

USAISEC

*US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300*

(4)

DTIC FILE

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES
(AIRMICS)

AD-A216 892

EXPERT SYSTEMS DEVELOPMENT METHODOLOGY

(ASQBG-A-89-033)

July, 1989

DTIC
ELECTE
JAN 19 1990
S B D

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800



DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

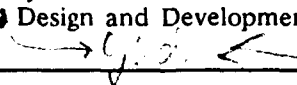
90 01 19 009

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704--188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION / AVAILABILITY OF REPORT N/A		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ASQBG-A-89-033			5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A		
6a. NAME OF PERFORMING ORGANIZATION AIRMICS		6b. OFFICE SYMBOL (if applicable) ASQBG - A		7a. NAME OF MONITORING ORGANIZATION N/A	
6c. ADDRESS (City, State, and ZIP Code) 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800				7b. ADDRESS (City, State, and Zip Code) N/A	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AIRMICS		8b. OFFICE SYMBOL (if applicable) ASQBG - A		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) 115 O'Keefe Bldg., Georgia Institute of Technology Atlanta, GA 30332-0800				10. SOURCE OF FUNDING NUMBERS	
				PROGRAM ELEMENT NO. 62783A	PROJECT NO. DY10
11. TITLE (Include Security Classification) Expert System Development Methodology (UNCLASSIFIED)					
12. PERSONAL AUTHOR(S) Dr. J. W. Gowens					
13a. TYPE OF REPORT		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) July 28, 1989	
15. PAGE COUNT 160					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Expert Systems, Artificial Intelligence, Knowledge Engineering, Expert System Sheels Design and Development 		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The purpose of this guide is to provide the procedures to develop expert systems (ES) that can be used locally to assist in mission accomplishment and to aid decision processes of the local command. Using the guidelines in this document, applications can be screened for expert system attributes, potential projects evaluated and initial estimates of the project made.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED / UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael J. Evans			22b. TELEPHONE (Include Area Code) (404) 894-3107		22c. OFFICE SYMBOL ASQBG - A

This research was sponsored by the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), the RDTE organization of the U.S. Army Information Systems Engineering Command (USAISEC). This effort was performed under Contract DAKF11-88-D-0011. This research report is not to be construed as an official Army position, unless so designated by other authorized documents. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.



THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

s/ James Gantt
 James Gantt, Chief
 Management Information
 Systems Division

s/ John R. Mitchell
 John R. Mitchell
 Director
 AIRMICS

The views, opinions, and/or findings contained in this guide are those of the authors and should not be construed as an official Department of the Army position, policy, of decision unless so designated by other documentation.

Table of Contents

Chapter 1	1
Introduction	1
1.1 Purpose and Scope.	1
1.1.1 Purpose.	1
1.1.2 Scope.	1
1.2 Background.	1
1.3 Format of the Guide.	3
1.4 How to Use This Guide.	5
 Chapter 2	 6
Fundamentals of Expert Systems	6
2.1 Introduction.	6
2.1.1 Definition.	6
2.1.2 Expert Systems.	6
2.2 Pre-History of Artificial Intelligence.	6
2.3 Modern History of Artificial Intelligence.	7
2.3.1 George Babbage.	7
2.3.3 Alan Turing.	8
2.3.4 John von Neumann.	9
2.3.5 Are Computers Intelligent?	9
2.3.7 The Dartmouth Conference.	10
2.4 Expert Systems.	14
2.4.1 The First Expert System.	14
2.4.2 The MYCIN Project.	14
2.5 Definition of Expert Systems.	18
2.6 This Guide.	25

Table of Contents

Chapter 3	27
Overview of the Knowledge Base	
Systems Development Methodology	27
3.1 Introduction	27
3.2 Traditional Software Development Model.	27
3.3 Iterative Development Model.	29
3.4 Detail Development Methodology.	31
3.4.2 Solution Definition.	33
3.4.3. Design and Build.	33
3.4.4 Testing and Transition.	36
3.5 Management Activities.	36
3.6 Route Maps.	38
Chapter 4	43
Project Initiation and Planning	43
4.1 Introduction.	43
4.2 Identification of Opportunities.	43
4.2.1 Motivation to Build Expert Systems.	44
4.2.2 The Role of the System.	46
4.3 Estimating the Project and Preparing the	
Workplan.	47
4.3.1 Estimating the Number of Rules, Frames and	
Objects.	48
4.3.2 Estimating the Effort for Project.	49
4.3.3 Preparing the Workplan.	50
4.3.4 Putting it all together in an example.	52
Chapter 5	55
Selection of the Development Team	55
5.1 Introduction.	55

Table of Contents

5.2	Selection of the Team.	56
5.2.1	Appointment of the Team.	56
5.2.2	Team Composition.	57
5.2.3	Selection of Knowledge Engineers.	57
5.2.4	Selection of stakeholders.	59
Chapter 6	64
Selection of the Expert System Shell	64
6.1	Introduction	64
6.2	Shell Selection Method.	65
6.3	Categorize your problem type.	65
6.3.1	Diagnostic Problems.	66
6.3.2	Monitoring Systems.	66
6.3.3	Design Problems.	67
6.3.4	Scheduling Problems.	67
6.3.5	Planning Problems.	68
6.4	Define Interface Requirements.	68
6.4.1	User interface.	68
6.4.2	Development Interface.	69
6.4.3	System Interfaces.	70
6.5	Define Platform Requirements.	71
6.5.1	Development Platform Requirements.	71
6.5.2	Delivery Platform Requirements.	71
6.6	Determine Shell Capabilities.	72
6.6.1	Classes of Shells.	72
6.6.2	Shell Capabilities.	74
6.6.3	Programming Techniques.	77
6.6.4	Vendor support and Cost.	81
6.7	Matching Requirements and Capabilities.	83

Table of Contents

Chapter 7	85
Knowledge Acquisition	85
7.1 Introduction	85
7.2 Documentation Review.	86
7.3 Methods of Interviewing Experts.	87
7.3.1 Nondirective Interviewing.	87
7.3.2 Structured Interviewing.	90
7.4 Expert Protocols.	94
7.4.1 Thinking Aloud Narratives.	94
7.4.2 Expert Systems Simulation.	96
 Chapter 8	 98
Knowledge Modelling	98
8.1 Introduction	98
8.2 Knowledge Modelling.	98
8.3 The Modelling Cycle.	105
 Chapter 9	 107
Design and Prototyping	107
9.1 Introduction.	107
9.2 Expert System Design.	107
9.2.1 User Perceived Objects.	107
9.2.2 Design Display Layouts.	109
9.2.3 Interaction Sequencing.	117
9.2.4 Output Behavior.	120
 9.3 Prototyping Phase.	 123
 Chapter 10	 125
Implementation Strategy	125

Table of Contents

10.1	Introduction.	125
10.2	Managing the Implementation.	125
10.3	Implementation Plan	128
10.3.1	Surface Implementation Actions.	128
10.3.2	Subsurface Implementation Actions.	129
Chapter 11	130
Documentation and Maintenance	130
11.1	Introduction.	130
11.2	Elements and Phases of Expert System Documentation.	131
11.3	Brief Description.	131
11.4	Functional description of an expert system.	137
11.5	The Proposal.	140
11.6	Requirements Definition Document.	142
11.7	Solution Definition.	144
11.8	Program Documentation.	145
11.8.1	Program Maintenance Manual.	146
11.8.2	Source Code -- Internal Documentation.	147
11.8.3	Documentation File.	148
11.9	User Documentation.	149
11.9.1	Intrinsic Documentation.	149
11.9.2	User Manual.	154
11.9.3	On-Line Documentation.	157

Chapter 1

Introduction

1.1 Purpose and Scope.

1.1.1 Purpose. The purpose of this guide is to provide the procedures to develop expert systems (ES) that can be used locally to assist in mission accomplishment and to aid decision processes of the local command. Using the guidelines given in this document, applications can be screened for expert system attributes, potential projects evaluated and initial estimates of the project made.

1.1.2 Scope. The guide has been written with the typical user assumed to be an officer or enlisted person with experience in microcomputers and some formal training in computer science, either in Army schools or a civilian educational institution. It can be used by commanders and/or staff personnel at all levels of command who wish to develop an ES to assist in the decision making process. These decisions might involve analyzing voluminous data from diagnostic sensors, analyzing data from readiness reports, analyzing risks for alternative courses of action, analyzing the consequences of several courses of action and other decision arenas where human judgment is a necessary element of the decision. Those personnel who are fully trained and totally familiar with computer operations and ES construction will find the guide a useful refresher and self check manual. The guide is designed to be used at all levels of ES development from hardware/software selection to coding of the system.

1.2 Background.

This guide was prepared under contract to the U.S. Army Institute for Research in Management Information, Communications, and

Computer Sciences (AIRMICS), Atlanta, Georgia. AIRMICS has been involved in decision making research activity that includes decision support systems (DSS), artificial intelligence (AI), and expert systems (ES) since 1977. This project was a continuation of that interest in the decision making processes.

The need for computer based decision assistance has been documented for some time. As technology shortened the time cycle for decision making in the military environment, it became apparent that automated support was necessary. Commanders today will have only hours or minutes to gather data, analyze that data, make a decision and issue orders.

Until recently, automation of decision making was beyond the reach of local commanders. The automation equipment required was too expensive and too fragile for deployment to lower levels of command. Today's powerful microcomputers have changed the face of computing at all levels of command. The latest microcomputer technology provides a hardware and software environment which is sufficiently powerful for the development of practical and useful ES. The next step is to provide the capability for local commanders to develop ES.

Nowhere is the need for speed in difficult decision making more apparent than in the Army's mobilization mission. Converting thousands of men and pieces of equipment from reserve to active status is a monumental task under the best of conditions. Performing mobilization under the duress of a world crisis is a task beyond comprehension. This current project is to develop a mobilization planning and execution expert system (MOBPLEX) to assist mobilization stations and FORSCOM J5 in the mobilization process both in the planning phases and the execution phases.

This guide provides a methodology to develop ES. The MOBPLEX is used as the basis for the guide. The process conducted to develop MOBPLEX is used throughout this guide as an example of ES development. However, the principles given in this guide are generally applicable to any ES development. This guide can be used by any commander or staff in the development of their specific ES.

1.3 Format of the Guide.

The guide consists of two volumes. Volume 1 is the Development Methodology and Volume 2 is an Evaluation Methodology containing methods for evaluation, validation and verification of expert systems. Validation and verification is an extremely difficult problem and is given separate treatment in this volume.

Volume 1 consists of 10 chapters as outlined below providing the details of the development methodology. An index is provided at the end of the volume to facilitate finding topics in this guide.

Chapter 1, Introduction, presents general information and motivation for this project.

Chapter 2, Fundamentals of Expert Systems, provides a basic introduction to the history and theory of AI and ES.

Chapter 3, Overview of the Methodology, presents a general synopsis of the methodology and an orientation to the remainder of the volume and its relationship to the methodology.

Chapter 4, Project Planning and Initiation gives criteria for selection of a candidate project with explicit criteria that can be applied to potential applications to test their applicability for ES development. In addition, the chapter gives guidelines for estimation of the effort in the project

and a work breakdown structure for designing the project plan.

Chapter 5, Selection of the Knowledge Engineering Team, is a framework for selection of the individuals to participate on the development team.

Chapter 6, Selection of the Expert System Shell, provides guidance on the parameters to be considered in the selection of the language (or expert system shell) in which to program the ES.

Chapter 7, Knowledge Acquisition, discusses the basics of knowledge engineering and knowledge acquisition techniques.

Chapter 8, Knowledge Modeling, provides a framework for developing a model of the data and knowledge required for the application.

Chapter 9, Design and Prototyping, discusses the problems of designing the user interface and other characteristics of the ES and the prototyping procedures.

Chapter 10, Implementation Planning, provides a paradigm for planning the implementation of the system so that it is accepted into the culture of the using organization.

Chapter 11, Documentation and Maintenance, provides assistance in defining documentation for the system and setting up the recurring maintenance activity for the installed system.

Volume 2 contains four chapters which are a collection of evaluation techniques that may be used in all phases of the development of expert systems. Evaluation begins with the selection of the project. Simple selection criteria were given in

Chapter 4 of Volume 1 on candidate selection. In Volume 2 more penetrating analysis is given for evaluation of the project from initiation through acceptance testing.

Chapter 1, Evaluation Overview, gives an outline of the volume and motivation for the volume.

Chapter 2, Impact Assessment, is a guide for conducting an impact assessment of potential applications that were selected using the criteria in Chapter 4 in Volume 1.

Chapter 3, Cost Benefit Analysis, provides guidance for preparation of a cost benefit analysis for the expert system. This is an important step in the decision to build an expert system.

Chapter 4, Knowledge Base Validation, presents procedures for the validation of the knowledge base in the expert system.

1.4 How to Use This Guide.

This guide is written in the order the steps to building an expert system are performed. Thus the chapter sequence gives the basic sequence of operations. On the other hand, the development methodology is one of staged, iterative development. As such, there is not a totally linear sequence of steps to be followed. The development team must build a plan for the project based on the needs of the targeted organization.

Chapter 3 provides the guidance on how to build a "route map" through the methodology. Thus, Chapter 3 is more than just an overview of the methodology. It is the keystone to the remainder of the guide. Understanding the "route map" and iterative development is the key to building a feasible work plan using the methods of this guide.

Chapter 2

Fundamentals of Expert Systems

2.1 Introduction.

2.1.1 Definition. Artificial Intelligence is a term that describes using sophisticated computer programs to embody advanced methods of computer technology. In many ways the term is misleading because it indicates that perhaps there is an element of human in the computer. Nothing could be further from the truth. Artificial intelligence, as it is commonly used and as it will be used in this guide, is merely software and/or hardware that draws from the current most advanced techniques of computer science. The issue of whether machines "think" in the sense that humans think is an unresolved philosophical question that we will not attempt to answer. If you are interested in this philosophical question, you are referred to McCorduck¹ as a good place to start.

2.1.2 Expert Systems. Our concern here will be with expert systems technology, a subset of artificial intelligence that holds great promise in making software more sophisticated in its operation as well as easier to use. These computer programs enable machines (computers) to do things that would require intelligence if done by people. In order to understand how expert systems became the complex study of computer programs that it is today, we need a reference to the historical beginnings of man's relationship with machinery and computers. Accordingly, this chapter will discuss the background and fundamental principles of expert systems.

¹ McCorduck, Pamela. Machines Who Think, W. H. Freeman, San Francisco, 1979.

2.2 Pre-History of Artificial Intelligence.

Man's fascination with machinery dates back to the beginning of civilized man. Some of man's earliest writings exhibit interest in machines that could have human-like qualities. These qualities were often captured in some form of robot, often manufactured from actual human parts, e.g. Dr. Frankenstein's monster. Other forms were mechanical devices to perform a function to aid man in his daily activities. One of the earliest forms of artificial intelligence was clock mechanisms. These "clocks" were mechanisms that could keep time but also perform a number of other functions. Pre-history is replete with examples of "clockworks", some real, some fanciful, that have been built or imagined over the centuries. McCorduck² provides an excellent overview of this pre-history period.

2.3 Modern History of Artificial Intelligence.

2.3.1 George Babbage. The modern history of artificial intelligence must begin with George Babbage in the early 19th century. Babbage created a device which he called the analytical engine and which had the capability of performing mathematical calculations very rapidly. Babbage is generally regarded to have built the first computer. Today, we would call his computer an analog computer. During Babbage's time, there was no clear distinction between what might be called intelligent thought and mathematical calculation. We now see arithmetic as a mere rote manipulation of numbers. But in Babbage's time it was seen to be far more fundamental to intelligence, probably because only the most highly educated and not even all college graduates could do arithmetic.

When Babbage made a machine that could perform calculations after some setup steps, the engine was seen as a severe threat to human

² McCorduck, op. cit.

intelligence. The Babbage analytical machine was a rather large contraption consisting primarily of hundreds of polished brass gears and rods that were stacked together in tall columns. The analytical engine was significant because it had the capability, at least in theory, to manipulate any symbol and not just numbers. This caused it to be quite different from previous calculators which did only simple arithmetic operations which had existed for several hundred years before the analytical engine. Babbage worked on various versions of the analytical engine for some 50 years and when he died in 1871 he had finished none of them.³

2.3.3 Alan Turing. Skipping forward five decades or so, the next figure in the pursuit of artificial intelligence was a British mathematician named Alan Turing. Turing took the next important step in the progress from simple clockwork mechanisms such as Babbage's to machines with intelligence. In the middle 1930's, Turing was considering the question of how to describe things that are computable. He decided that whatever could be expressed by numbers and calculated by a machine could be defined as computable. Certainly performing arithmetic operations were computable. But other operations such as writing good poetry were severely in question as to whether they were computable.

In order to work with this problem, Turing created what he called the universal machine. These machines were entirely theoretical at the time he created them. The machines that Turing was talking about were machines that manipulate information in contrast to machines that performed other functions. Actually, it was not possible to build a universal machine at the time Turing invented it. But he was able to prove such a machine could exist.

³ Lady Ada Lovelace was a contemporary of George Babbage. She studied with Babbage and wrote a popular essay explaining the analytical engine. She is sometimes credited with inventing computer languages. The Ada computer language is named in her honor.

Turing began work on building a physical machine in the middle 1940's. Simultaneously, several teams in the United States were also building computers. All the machines that were being worked on at that time followed the principles that Turing proved in his universal machines.

2.3.4. John von Neumann. The next major figure in the history of computing is an American mathematician originally from Hungary by the name of John von Neumann. Von Neumann followed Turing's early work in computing, but added to it the study of the physiology of the human brain. He hoped this study would enable him to build a computer with human thinking capacities.

Many of the individuals working on computers in the late 40's and early 50's had envisioned that a computer would eventually be a mechanical and mathematical model of the human nervous system. Neumann was trying to create what might be described as an electronic nervous system. He believed that the design of the computer using, first, vacuum tubes, and, later, transistors could be built in a way to create a neural network much like the network of neurons in the brain and that the computers might operate very much like the human brain. Von Neumann was not alone in his beliefs. Most scientists working in the area of this fledgling science known as computer science held similar beliefs. It is probably from this early work in attempting to emulate the human nervous system that the foundations for calling advanced computer technology "artificial intelligence" were solidified.

2.3.5. Are Computers Intelligent? At this time, scientists saw the stirrings of life inside their computers. This prompted Alan Turing to create a test now known as the Turing test which could be used to determine if the computer was "intelligent" or not. This test involves the placing of a human being in front of two

teletypes. The human being asks questions of the two teletypes - teletype A and teletype B -- without knowledge of whether the teletype is connected to a computer or if there's another human being responding from the teletype. Turing stated that if the individual asking questions and communicating with the two teletypes was unable to determine whether or not the teletype responses were coming from a computer or from a human then we would be able to define that computer as being intelligent. Turing and many of his colleagues at the time believed that the creation of such a computer was just around the proverbial corner and that surely by the end of this century computers of this type would be generally available. We now know as we are about to enter the last decade of this century that Alan Turing and his colleagues were highly optimistic.

By the beginning of 1950, digital computers were beginning their first era. At that time it was clear that very powerful computers could be built. They certainly did not approach the complexity of the brain, but on the other hand, they were powerful enough to attempt intelligent tasks with them. The field of artificial intelligence really begins just at this point. Digital computers were made and they were waiting. Empty machines with tremendous computing power waiting to be put to work.

2.3.7 The Dartmouth Conference. The next important step in artificial intelligence history was the Dartmouth Conference put together by John McCarthy with assistance from several others. McCarthy took his undergraduate degree at Cal Tech and completed his Ph.D. in mathematics at Princeton in 1951. During his graduate education, McCarthy studied under von Neumann at Princeton and began to think about the theory of computers and their relationship to problem solving. It was here at Princeton that he first began to experiment directly with computers.

In 1956, McCarthy joined with three others who thought it might be a good idea to get all the individuals together who were interested in mechanical intelligence. McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon proposed a summer conference to be held at Dartmouth College to the Rockefeller Foundation. McCarthy was the prime mover behind this conference and coined the term artificial intelligence in his proposal. The Rockefeller Foundation provided \$7500 for the conference and all the significant characters in the field attended: the proposal group, Herbert Simon, Alan Newell, Arthur Samuel, Oliver Selfridge and Ray Solomonoff. All these individuals became the founding fathers of the entire field of artificial intelligence.

Herbert Simon and Alan Newell from Carnegie Mellon University were further along than anyone in the conference. These two men had come to the conference with a computer program that they called the Logic Theory Machine. The program had as its purpose to solve problems in logic in the same way that humans might do. It had a pretty good number of capabilities. It could perform substitutions of one expression for another, break up a problem into a series of sub-problems, and it could decide which sub-problems to work on using a method they called chaining. The program could actually find proofs for 38 of 52 theorems it tried in the Principia Mathematica, the epic work of mathematical logic by Alfred North Whitehead and Bertrand Russell. The program therefore could solve problems pretty close to the level of a college student and it found one proof more elegant than the version offered by Whitehead and Russell.

Another pioneer of artificial intelligence, Arthur Samuel, was at the Dartmouth Conference. Samuel was talking about a computer program which he had written to play checkers. Samuel was an IBM employee and helped to build the first commercial computers for IBM. He utilized the same computers that were being built for

commercial applications. He would work on that at night in order not to disturb the business operations out on the factory floor during the day.

There was also a great interest in chess. McCarthy himself was interested in the idea of computer chess. And it was at this time that McCarthy contributed a great insight was gained into the strategy of problem solving using computers. Because chess has so many possible moves, it's not possible to look ahead to all the possible moves that can be made. Therefore, some method had to be designed to determine how many moves ahead a chess playing program might look and the strategy to use for looking ahead.

McCarthy began work on utilizing human-like problem solution search characteristics to determine how to look ahead in chess-playing programs. He created a mathematical method which he called the alpha-beta heuristic. First you look for the most promising move that you can make, then you look for the most damaging thing your opponent can respond with. Then you stop your search on all lines that give your opponents a powerful answer. In this way you can reduce the number of possibilities that you have to consider when you search for a solution. This principle has become an important principle in the area of artificial intelligence known as expert systems today.

It was about this time in the middle 1950s that high level computer languages were invented. Prior to this, computer programming was a tedious process of entering zeros and ones into the computer. In the middle and late 1950s two of the most important programming languages ever invented COBOL and FORTRAN were invented. To handle the special characteristics of artificial intelligence work, John McCarthy invented a language called LISP. LISP has become known as the language of artificial intelligence because it was created to perform symbolic manipulations in a very specialized kind of

way. LISP is generally regarded as being the second computer language to be written. FORTRAN having been invented in 1956 and LISP in about 1958.

2.3.8. In the 1960's there was a flush of hope that the creation of a highly intelligent computer was just around the corner. Many programs were written that could perform many human like operations. Computer programs were playing championship checkers, pretty passable games of chess and many other things. Programs were choosing portfolios of stocks and doing almost as well as human stock investors. There was even a program that could solve integrals from freshmen calculus and programs that could solve arithmetic word problems at the six grade level.

This excitement became a public excitement and many of the researchers became very bold in their predictions. Herbert Simon wrote in 1957, "It's not my aim to surprise or shock you but the simplest way I can summarize is to say that there is now in the world machines that think, learn and create. Moreover, their ability to do these things is going to increase rapidly until in a visible future the range of problems they can handle will be co-extensive with the range to which the human mind has been applied...a digital computer will be the world's chess champion within 10 years. A digital computer will discover and prove an important new mathematical theorem. And theories and psychology will take the form of computer programs."

Another researcher said that in three to eight years we will have a machine with a general intelligence of an average human being.

This optimism grew a crowd of both critics and supporters. The ancient debate between man and machine grew up again. The debate was partially philosophical in the determination of how one can define intelligence. Can a machine ever be intelligent? Actually

the predictions of the researchers came to nothing. Most of the expected advances failed to occur. The performance of programs that had been written began to look less and less impressive. While we haven't given up on some of these goals for computer programs, we now know that computers will likely never be as intelligent as a human for many operations.

2.4 Expert Systems.

2.4.1 The First Expert System. The next step in the path toward expert systems came from a individual who had been a graduate student under Herbert Simon at Carnegie Mellon. His name was Edward Feigenbaum. Feigenbaum worked with Simon at Carnegie Mellon and later moved to Berkeley, California. There he joined forces with Julian Feldman and they went to work in the fledgling field of artificial intelligence. Feigenbaum had a degree in cognitive psychology and was interested in creating structures that would simulate the ability to learn from a human being. Feigenbaum and Feldman published one of the first books in artificial intelligence which was the collection of scientific papers called Computers and Thought. It was published in 1963 by McGraw-Hill.

Later Feigenbaum moved to Stanford and there he began work with a Nobel laureate Joshua Lederberg who had won the noble prize in genetics. They began work on a program called DENDRAL. DENDRAL is a system to define adequate hypothesis to explain data that came out of mass spectrographs. Feigenbaum did this by developing a computer program which captured the knowledge and expertise of the scientist working in the area. DENDRAL, therefore, was the precursor of the next generation of AI programs which have come to be called expert system or expert systems.

2.4.2 The MYCIN Project. The next step in expert systems technologies occurred at Stanford University in a project called MYCIN. The MYCIN project was to build a expert system that would

have the ability to diagnose infectious blood diseases in the same way that a physician might perform the diagnosis. This project was conducted by William J. Clancey, Bruce Buchanan and E.H. Shortliff. These individuals created the most powerful expert system known up until that time. MYCIN is an interactive program that simulates a medical consultant specializing in infectious diseases. It engages in question and answer conversations, (lasting 20 minutes on average) with doctors needing specialist help. The physician asks MYCIN for advice on the identification of microorganisms and the prescription of antibiotic drugs and also for explanations of its advice expressed at the appropriate level of detail.

MYCIN's explanatory capability was not put in merely for show. It enabled the physician to rationally decide to either reject or accept the program's advice; it allows non-specialist doctors to learn about infectious blood diseases; and it allows human consultants to make improvements in the program. The program itself was designed as a program providing advice and physician support rather than having the complete knowledge to give complete diagnosis. The MYCIN program gave the physician the capability of asking how a particular diagnosis was determined where upon the program would respond with all of its decision criteria and decision points or the physician could ask the machine why it was requesting a particular piece of information. And the machine would respond with its reasons for needing the information. The question and answer dialogue, the why and how explanation facilities had become hallmarks of all future expert systems.

MYCIN evolved into a program known as EMYCIN. EMYCIN was a program that utilized all of the driving mechanisms of the original MYCIN program but separated out the driving part of the software from the knowledge contained in the software. EMYCIN was the first expert system program with the capability of having a new expert system placed into the system without having to change the user interface

of the driving portion of the software. This was, therefore, the first so called expert system shell.

The next program of significance was written by John McDermott at Carnegie Mellon University. McDermott was hired by the Digital Equipment Corporation to build a system that would configure VAX computers. This system began to be known as XCON. XCON was an expert system built very much like the MYCIN system but with the capability in knowledge to determine which pieces of hardware, cables and so forth were required when a particular model of VAX was ordered. This was a highly complex process and the XCON program provided considerable savings to Digital Equipment Corporation in their ability to deliver VAX computers. John McDermott wrote the program beginning in the Fall of 1978 and it has been expanded many times since then. The configuration effort involves insuring that an order can be built and that all necessary component parts had been included in the order as well as determining just what part types, cable lengths, etc. were required for successful assembly. A typical order would deal with 50 to 250 components and 25 to 125 pieces of information about each component. It typically takes a thousand steps to determine exactly what components should be included should be included in the final package that is assembled and delivered to the customer.

Prior to the development of XCON, technical editors in manufacturing reviewed all customer orders for technical correctness and for completeness. This review required 20 to 30 minutes per order and still frequently resulted in systems arriving at the customer sites without all the necessary components.

DEC had made three efforts to develop a configuration program using conventional programming techniques and failed. Both because the knowledge involved in the tasks proved very difficult to define and because the components and assembly requirements kept changing.

The task is highly conditional. Typical configuration involved 1000 steps and there are on average three possible paths that can be followed at each step. In order to write this system, McCarthy created a new kind of programming language now known as Ops 5. It is a knowledge base or ruled base system and currently contains about 6,000 rules. It allows the VAX configurations to be created in approximately four minutes. DEC has been so successful with XCON that it has created numerous other expert systems internally to be used for other tasks independent of XCON. In fact, Digital Equipment Corporation may be one of the largest users of expert systems in the world. Internally the DEC organization has more than 300 people involved in AI that are in four different groups in the organization. This is an indication of how DEC perceives AI to be as strategic section of their company. DEC truly perceives its corporation as a knowledge network.

Since 1980 artificial intelligence and expert systems has been growing exponentially. One of the major factors in the growth of expert systems has been hardware development. In the middle 1950's at the very birth of AI, hardware was large very slow and extremely expensive. In 1975 when MYCIN was being written, computers to run expert systems probably cost mostly in the million dollar range. By 1980, special purpose computers had been built to run expert system programs and to perform artificial intelligence tasks. However, these pieces of hardware were still in the \$100,000 to \$300,000 range. In addition to that the language that most of these programs were written in LISP was relatively difficult to interface to conventional computers and to convention programming languages. This was simply a characteristic of LISP. It had nothing to do with the hardware. For this reason, most of the business oriented application scientist must preferred to utilize a conventional language and conventional hardware to produce their expert systems.

In the middle and late 1980s, the hardware and language capabilities have increased to the extent that we can now perform expert systems in artificial intelligence research utilizing conventional languages and conventional hardware. The power of the PC has become so great that it far exceeds that of most of the special purpose hardware of the 1970s. For this reason, expert system programming and knowledge base systems have become feasible for distribution in all areas of the Army environment. The current focus of expert system is on the identification of tasks to be computerized that are above the clerical level and for which the business objective can be evaluated positively.

2.5 Definition of Expert Systems.

2.5.1 This section will contrast transaction based processing and expert system processing. Every computer program contains within it some degree of expertise. It must be obvious that computer programs perform tasks we can consider to be reasonably intelligent. In transaction based programs, such as an accounting program or an inventory management program, the system knows what accounts to post if it is an accounting program or it knows when to reorder inventory if it is an inventory program. These activities are manifestations of the program's expertise. Then, how are transaction based programs different from expert system programs? The difference is in a expert system program three things are explicitly defined which are unique:

- 1) The knowledge is very transparent to the user. That is to say when you look at a expert system program the user of that program says, "That program is doing something smart. It is doing something intelligent and I can see that intelligence as it acts." In a transaction based program, typically the expertise is not transparent to the user. In fact, the user may well be confused by the computer program.

2) The second major difference between transaction based programs and expert system programs is in an knowledge based program the driving software is separate from the expertise of the system. There are two pieces to this program that are totally and distinctively separate. In an transaction based system the expertise and the driving software for the program are contained in a single program. It is extremely difficult to see expertise in the program source code and distinguish it from execution sequence and overhead statements. They are all intermingled together. There is no separation.

3) The third difference between these types of software is that expert system programs can explain how and why they have taken a specific action. This is one of its most distinguishing features.

These three differences are the characteristics we use to distinguish expert system from transaction based systems. Expert systems are also often called knowledge based systems.

2.5.2. When we refer to an expert system, we will be referring to a class of software that is knowledge based. It has a separate driving program from its knowledge. And it has knowledge that the user perceives as being transparent. This means that when we look at the knowledge we can understand it without special training.

2.5.3. An expert system is a software program. What we are doing is not black magic. We're not creating something where nothing existed before. We are simply writing software. And we're learning how to write software in a more clever way so that we can encode behavior of a specific expert from a narrow domain into that program. We can encapsulate that knowledge so that other users, people who may not be quite so expert, will have access to the best expert's expertise in our organization.

Another distinctive aspect of the software is that it can explain its behavior. It knows why and how it has reached a conclusion. This is another glaring difference between traditional software and expert system software.

2.5.4. What makes these pieces of software special? There are a lot of different things that makes them special. But one of the most important ones is the dynamic ordering of solution paths. When we create software in a traditional sense the solution paths to every problem are defined in the code. Computer programs operate on the premise that each instruction is performed sequentially unless the order is interrupted by branching instructions.

This is completely different from the way an expert system processes information. An expert system processes information based upon data relative to the case under consideration. The solution order of the problem is determined by the data of the case. The solution order is therefore completely data driven. This may be compared to table driven COBOL, where parameterization of the program occurred as a result of data entering the system. In expert systems, that idea is carried to an extreme so that the entire program operates that way. Therefore the software itself is simply a place holder. It contains generic knowledge about the domain. When case information enters the program, it knows what to do based on the facts of the case.

2.5.5. There are a lot of uses for expert systems. They are in use in all industries: military, telecommunications, computer companies, CPA firms, insurance companies, financial services organizations, banks. Almost every topic that you name has an aspect of expertise to be emulated in expert systems. Modern expert systems draw from academic disciplines of all types:

psychology and the social sciences, philosophy, management sciences and mathematics as well as engineering and computer sciences. All of these disciplines come together to form what we know today as the field of expert systems.

2.5.6. An expert system shell is a generic piece of software that composes the driving software of the expert system, but which has no knowledge embedded in it. The difference between an expert system and a transaction based system is that the driving software and the expertise are in separate modules. The driving part of the software is called the shell. There are many of these which are commercially available ranging from very large and expensive to quite small and moderate in price. The numerous shells on the market make selection of the "right shell" difficult but, the selection of a shell is a topic for another discussion. Let's note here though that there are substantial differences in shells. A shell is most useful for the prototyping phases. Later, a special purpose shell is often created for specific applications.

The architecture of an expert system may be conceptualized as two large boxes showing the division between the driving software and the expertise as shown in Figure 2-1. The inference engine is nothing more than an interpreter. It has the ability to interpret and order rules and facts to solve problems. It's an interpreter in the strictest sense just like you might have an interpreter in a computer language. As facts are input to the system, they activate rule clauses (we will look at a rule in a moment) as the facts match the clauses. The pattern matching process is the essence of the operation of the inference engine.

2.5.7. The rule base contains the expertise of the expert system. This is the domain knowledge. Rules are generic in the sense that they do not refer to any specific case, but rather they hold the

Expert Systems: Architecture

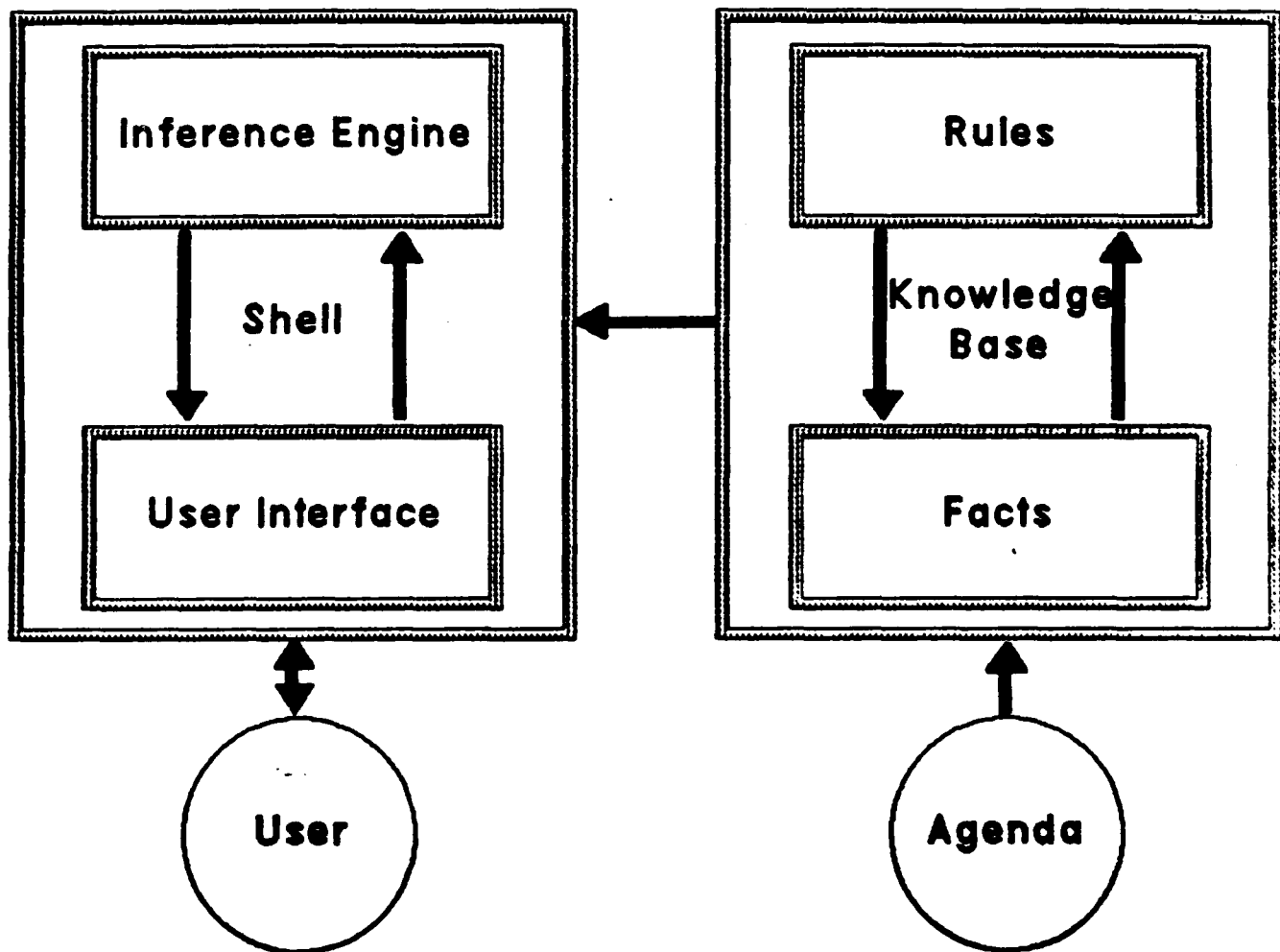


Figure 2-1. Expert System Architecture

domain knowledge for all possible cases that might be analyzed by the system. In this sense then they are place holders in much the same sense that a variable name would be in a traditional program. They are much more complex place holders when compared to variables in that they may have a number of clauses. These clauses may themselves contain variables. In larger shells, the rules form a rather complex programming language. A rule may look like the following:

```
IF (personnel fill is high)
    and (equipment fill is high)
    and (basic load is on-hand)
    and (unit training is complete)
    and (equipment is packed)
THEN (unit is combat ready)
    and (unit is ready for port call)
    and (request port call from MTMC)
```

2.5.8. The fact base is the collection of attribute values that describe the case that you're studying. In other words, this is the input. It comes from the user. It may be information which comes from a human user in a question and answer dialogue or it may come from a machine user such as a computer database, as we discussed under user interfaces. As we said earlier, rules are place holders, so once facts enter the system, they find the rule clauses which match. When they match with a rule clause, they activate that clause of the rule. When all conditional clauses of a rule have been matched, we say the rule fires.

2.5.8.1. There are several major ways to represent knowledge (or expertise) in an expert system. The most commonly used are: rules

and objects (frames)⁴. Even though the expertise. in the "rule base" also be represented in "objects," we will typically still call it a rule base. More correctly it should be called a knowledge base, technically this includes the fact base (see below.)

Rules are methods of organizing expertise in a series of IF/THEN statements as shown above. Objects are a way of organizing knowledge in a "frame" of attributes that describes the object with emphasis on default values. An object is a collection of attributes that define an entity by listing its characteristics. Objects usually are defined hierarchically, top down, with class, sub-class, and instances. Objects are connected with directed arrows to form a semantic network. The semantic network defines inheritance properties for the objects. In this method, lower objects need only to be described by differences from their parent because the parent characteristics are "inherited" to the child. These inherited properties are the default values for the child object. For example, we could define the class object of mammal, a sub-class object dog and instance of a specific dog of Buzzy. The properties defined in mammal do not need to be specifically defined for Buzzy, because they are inherited through the semantic network from the class mammals.

2.5.9. When facts enter the system they activate rule clauses. When all clauses of a rule are activated, the rule fires which activates the conclusion part of the rule. This conclusion will either activate further rules or give output to the user. The process of activating one rule then another is called chaining because the inference engine is chaining from one rule to the next. Chaining may be either forward or backward, depending on the way

⁴ An object and a frame are essentially the same thing. In this guide, we will use "object." Either could be used interchangeably in most cases.

the inference engine is written. Forward chaining and backward chaining are not equally simple to implement in software. Backward chaining is easier and is the usual type of chaining in the cheaper PC shells. In addition, certain classes of problems are more appropriate for forward chaining and others are more appropriate for backward chaining.

2.5.10. A simple example of how an expert system works is to consider a truck mechanic. Suppose you take your HUMV to the motor pool for repairs. The reason you go to that individual is because you have expectations that he has experience in repair of HUMV's which will enable him to diagnose the problem. This experience is his expertise and corresponds to rules in the knowledge base. When you give him the symptoms of the way your HUMV is acting, you are providing the facts the mechanic will use to solve your problem. He can then make the appropriate repairs to the vehicle. This is precisely the way an expert system works. In fact, expert systems were designed to emulate this type of problem solving human behavior.

2.6 This Guide.

The remainder of this guide will be to provide a framework and flexible methodology for the develop of expert/knowledge base systems. We want to give you a grab-bag of chosen techniques that you can utilize in the Army environment to develop and utilize expert systems effectively. It is our hope that you will see this as a new way of thinking about problem-solving and a new way of programming. In fact, we must keep in mind that all computer programs contain knowledge and expertise. The way that knowledge and expertise is encoded into the computer program is the issue. Expert system programs, encode knowledge in a rule format that allow programs to process symbols more than numbers. The knowledge is captured in rules that are reasonably easy for a human being to read and to understand, in contrast to the complex if/then

statements that would be in FORTRAN or COBOL program. Characteristics of expert system programs, in summary, are that they are the separated driving mechanisms or inference engine from the knowledge base and they have the capability to explain themselves with both how and why responses and explain mechanisms.

Chapter 3

Overview of the Knowledge Base Systems Development Methodology

3.1 Introduction

The methodology for development of expert systems is somewhat different from the system development life cycles for traditional transaction based processing system. In the expert system methodology, processes replace phases and stages and during system development dynamic activation of processes allows the system to evolve through a series of iterations. Each iteration is a well-defined process, but the number of cycles or iterations through these processes is determined only by the developed system, and/or the evaluation taking place through the development.

3.2 Traditional Software Development Model.

The cornerstone of software development is the concept of software development methodology. The traditional (waterfall) software life cycle development model is shown in Figure 3-1. This model defines the basic phases of a software development project and describes the output of each phase. Closely associated with the life cycle model is an accompanying project management technique that relies on reviews at the completion of each stage to establish a true status of the development project.

The use of the life cycle model has helped to relieve the software crisis. But there are still rather severe fundamental problems. First of all, the traditional life cycle model is linear. The use of this traditional life cycle model assumes that each step can be completely and correctly implemented before moving to the next. Specifically, it makes the assumption that one can completely understand all of the systems requirements and that these can be described and derived at the beginning of the project.

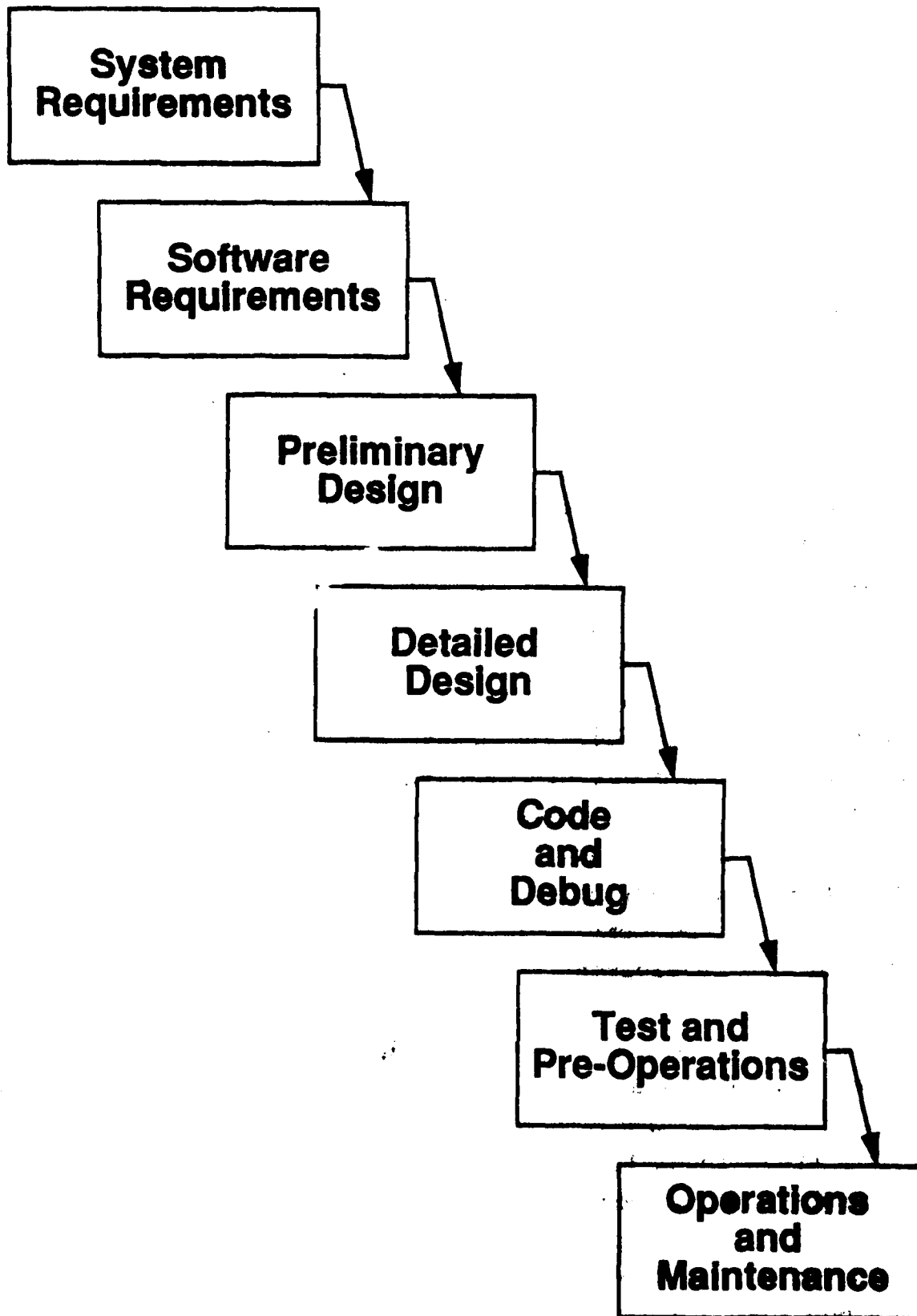


Figure 3-1. The Waterfall Model of Software Development

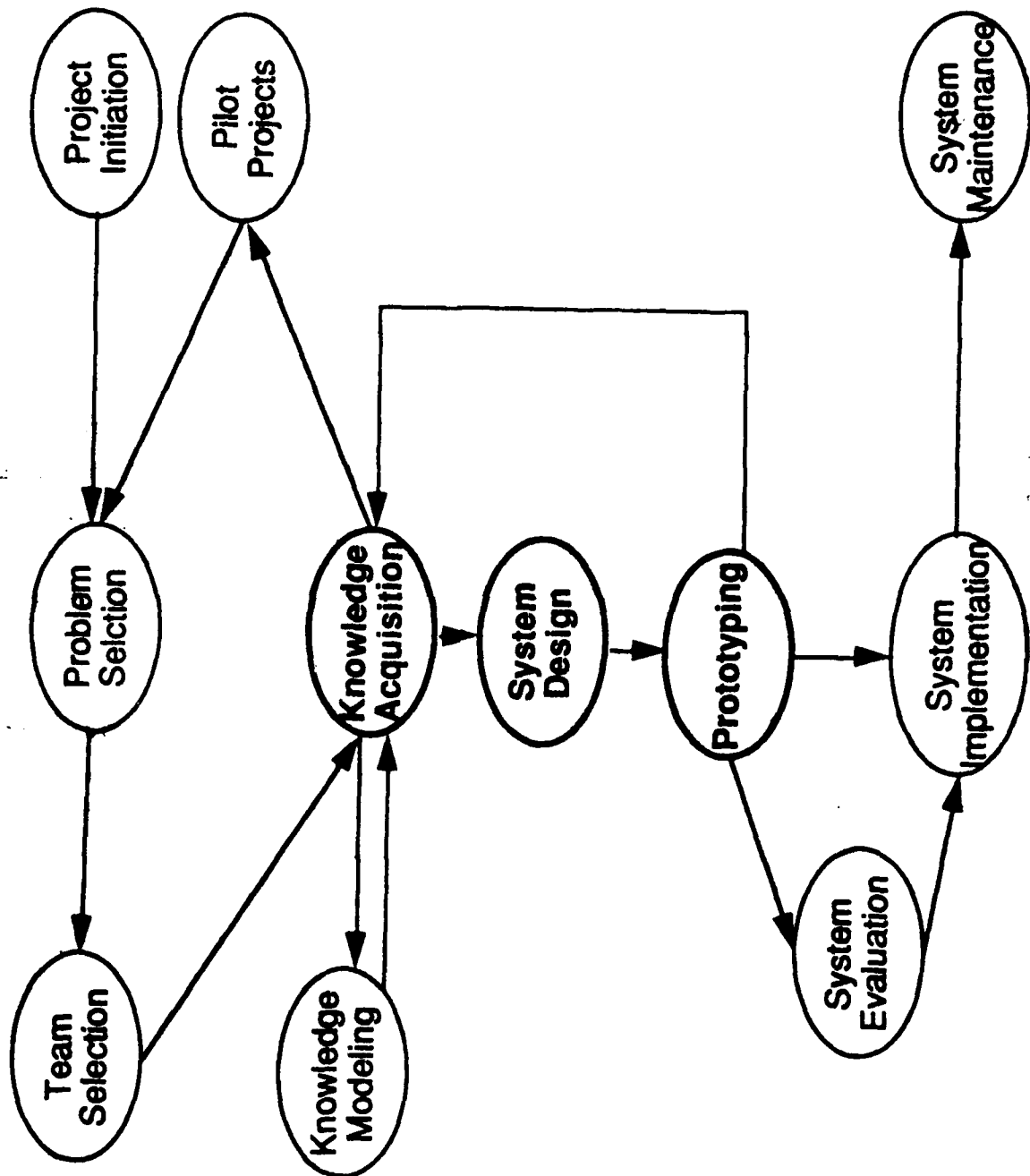
Unfortunately this assumption is invariably incorrect, because it is very difficult for either the user of the software or the developer of the software to know precisely what the full requirements of the software will be. In most cases, some portion of software must be developed and utilized before anyone can really understand what needs to be done.

3.3 Iterative Development Model.

The expert systems development life cycle is based on a principal of iteration of software development. Iteration is inevitable in any large software development project; but it's difficult to control this iteration when the linear standard software development life cycle model is used. The expert systems development methodology is specifically designed around the assumption that requirements definition cannot be done in the beginning of the project but rather must be an ongoing process throughout the development project. Therefore, in expert systems development we have consciously chosen to utilize a development life cycle that is nonlinear in nature. Figure 3-2 provides an overview of the development methodology utilized in expert systems development.

Observe that the methodology in Figure 3-2 is a nonlinear process and is designed for iteration through various stages many times. The methodology is based on the premise that the full design of the software can only be accomplished by a design-build-test cycle with feedback from users in order to build a system that is fully responsive to the requirements. In this way, the project can be controlled to reduce both technical risk and cost risk since the direction of the project can be shifted during any of the numerous review periods in the cycle.

For example, in the problem definition and selection stages, an organization could cycle through the processes of problem



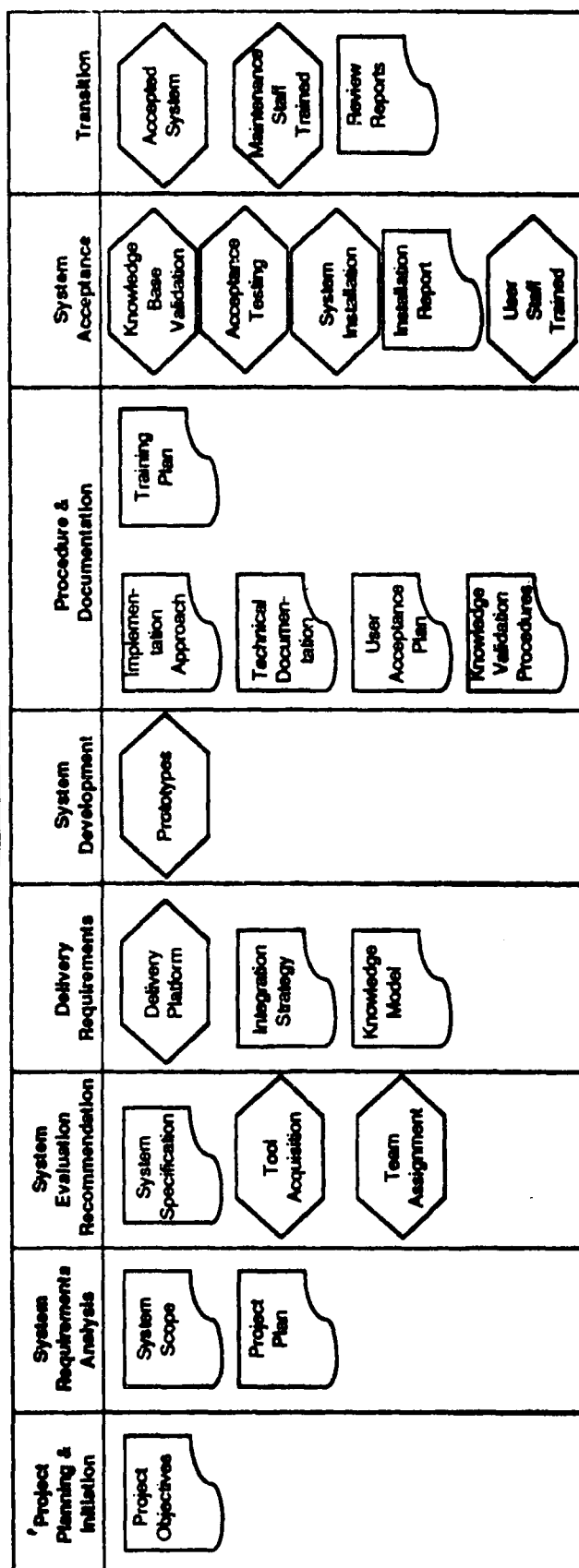
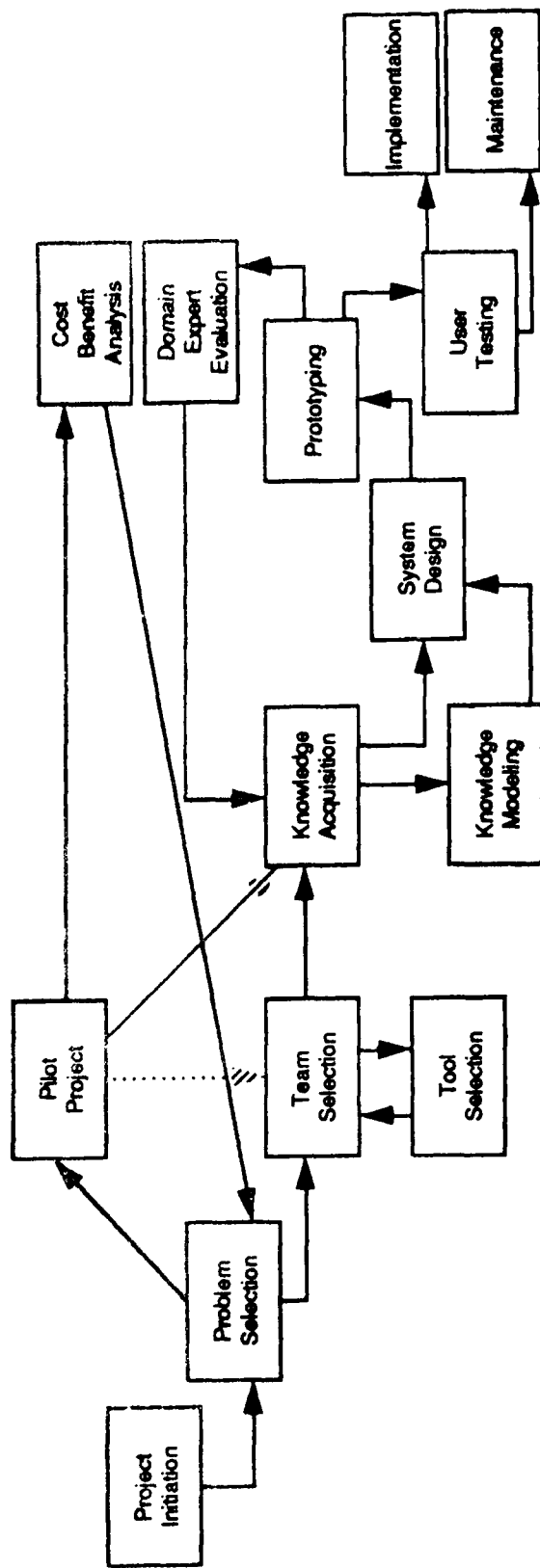
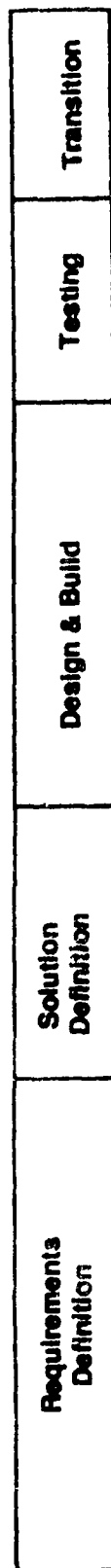
selection, team selection and pilot projects several times, perhaps with competing projects, until the particular project is selected. Similarly, it is anticipated that the subcycle of knowledge acquisition-design-prototype will be repeated several times in every expert systems project. This interactive nature of the development methodology is what gives it the tremendous power to produce implementable systems that satisfy users.

3.4 Detail Development Methodology.

The development methodology shown in Figure 3-2 has been operationalized as shown in Figure 3-3. In this figure the development methodology is shown as consisting of five basic steps: requirements definition, solution definition, design and build, test, and transition. These five stages are the five stages of any system implementation methodology; however, the differences are not in the stages, but how they are implemented.

3.4.1 Requirements definition. In the requirements definition phase several activities must take place. The project initiation phase (refer to Figure 3-3) is a preliminary phase to provide the motivation for the need of a system. This is a nebulous phase when the project is in its embryonic stages. The deliverable from project initiation is the project objective. This is a one page description of the goals of the project and the problem it addresses. In some cases several competing project descriptions might be written and evaluated before the final decision is reached on the project.

Problem selection is the definition of the scope and the project plan. This includes selection of candidate problems areas, estimating the project and preparation of the work plan. This phase can also include team selection, tool selection, and pilot projects. A pilot project, which is simply a very small development of a potential system, may or may not include a small



Key: Documentary Deliverable Non-documentary Deliverable

Figure 3-3 Implementation of Development Methodologies

amount of knowledge acquisition. Evaluating competing project and determining the cost benefit of a project will also be conducted during this phase. Documentary deliverables during this phase include: the system scope and the project plan. Additionally, there may also be a systems specification. The team for development of the system should be assigned during this phase and the tool acquired if necessary.

Because of the iterative nature of the development cycle, the requirements definition phase overlaps into the solution definition phase. As specifications are being prepared, solution activities will be in progress. In addition, repetition of requirements definition might be necessary after results of analysis or pilot projects are complete.

3.4.2 Solution Definition. The solution definition phase provides direction in designing a system which can be integrated into the using organization's environment. To be sure, solution definition overlaps problem selection, team selection, and pilot project phases; but solution definition's primary purpose is to formulate the integration strategy and to select a delivery platform. This will require understanding of the knowledge environment. This phase usually includes a small amount of knowledge acquisition and preparation of a preliminary knowledge model. The knowledge model will be modified during the design and build phase, and it is necessary here only to prepare the model sufficiently to find the delivery solution. Deliverables from this phase include: a decision on integration strategy, delivery platform, and the preliminary knowledge model.

3.4.3. Design and Build. There are four major steps in the design and build phases: knowledge acquisition, knowledge modeling, design and prototyping. In knowledge acquisition, the knowledge of an expert is discovered and placed into a form that can be put into

the software. In attempting to understand knowledge from an expert, the process will often also include knowledge modeling. The knowledge modeling procedure is understanding the underlying structure of the knowledge from an expert and it often takes the form of diagram that can be interpreted to show the knowledge flows and the process that an expert goes through in his decision making process.

Knowledge acquisition and knowledge modeling lead to the next step, system design. System design is the process of determining the actual code modules that will be utilized to produce the expert system. Of primary importance in this phase is the design of the user interface. The design of the user interface will often be the difference between a system that is accepted and one that is rejected. The interface design will also often determine the code modules breakdown for the system. The code module breakdown form natural work units that can be worked on by several programmers in the team or sequentially by a single programmer.

The actual programming of the code modules is the prototyping phase. In the prototyping phase, we intertwine the process of system design and programming with feedback to the knowledge acquisition phase. In the prototyping phase, a system evolves interactively between the knowledge engineer and the domain expert by creating a portion of the program, allowing the expert to evaluate it, and then using the evaluation from the expert to suggest the direction of the next phase in the prototype. During this evolution process, the prototype may undergo drastic revisions as the problem is better understood by the builder and the knowledge acquisition process is better understood by the domain expert.

This new approach does not eliminate the need for structured programming techniques, however some application require more

design process and may be more of a design problem than an implementation problem. Expert systems are certainly in this category. These design problems require that programming systems allow the design to emerge on the basis of experimentation with the program so that in effect the design program and program develop together. A prototyping approach to programming amplifies the programmer in the interest of maximizing his effectiveness while solving these design problems. In effect, the prototype becomes a dynamic specification of the system and changes rapidly as the domain expert and the knowledge engineer understand the problem and themselves better.

One of the reasons why this prototyping methodology is required is that experts or other users do not always understand exactly what they want or how they perform certain tasks. Experts for example will tell you that they perform a task in a certain way when in reality they do it another way. The reason is that they often times do not themselves understand exactly how they are performing a knowledge intensive task. This creates a "knowledge acquisition bottle neck" because the success of the knowledge elicitation from an expert determines the effectiveness of the system.

The knowledge acquisition bottleneck is also the reason for failure of many traditional systems. In traditional systems, this failure will often go under the guise of "inadequate or improper requirements specifications." Many a programmer has been heard to say "Why didn't you tell me that in the beginning?"; only to hear the answer "You didn't ask me." The iterative development is designed to reduce the frequency of occurrence of these kinds of misunderstandings by allowing the program to evolve through a series of iterative stages, rather than attempting to completely specify all the requirements for the program in a requirements definition at the beginning of the project.

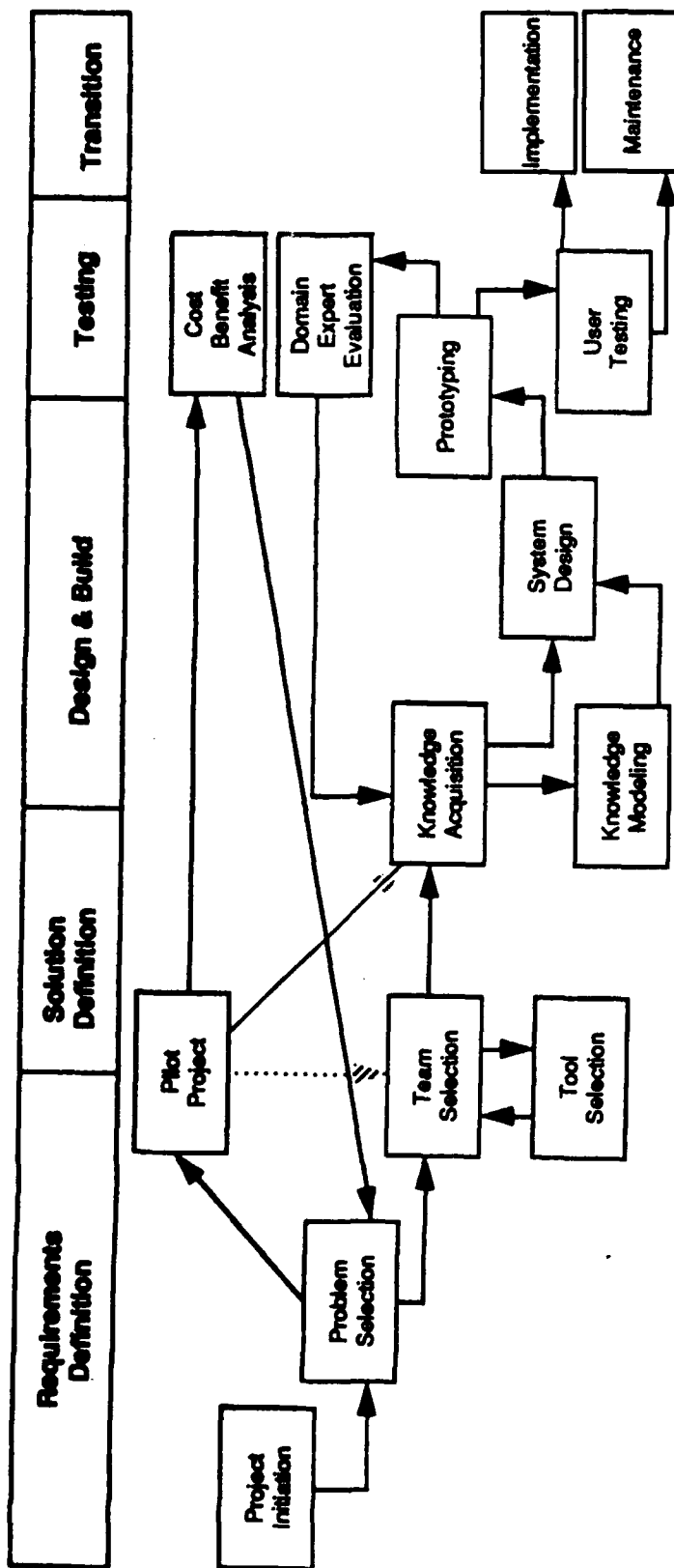
Deliverables during this phase of the project are the prototypes that are prepared by the knowledge engineer and the expert. As the prototype begins to home in on its system to be implemented, consideration must be given to the implementation of the system.

3.4.4 Testing and Transition. Implementation of the system requires thorough user testing and evaluation before the system is released. This user testing takes several forms. At the same time, experts should also be evaluating the system to determining its validity with respect to its problem solving. User testing and expert knowledge based validation are discussed in the evaluation methodology in volume 2 of this series. During this phase, however, several deliverables are required. An implementation plan and a user acceptance plan as well as a knowledge based validation procedure are required in order to validate the system.

Technical documentation, programmer as well as user documentation, will be required. During this phase a training plan should also be developed to train both users on the system as well as maintenance staff. At the conclusion of testing an acceptance will be required. At this point the knowledge base for the system should be validated, acceptance user testing should be completed, and the system should be installed, deliverables here include an installation report and various training activities to train the user staff and the maintenance staff. The last step in the transition phase is a final report and overview of the final accepted system. This report is reviewed by management of the builder and user groups.

3.5 Management Activities.

Figure 3-4 shows some of the management and quality assurance actions that are required during the development cycle. Management is required to perform both formal and informal reviews of the system as it's being constructed. Informal reviews include the



Management Action

Quality Assurance Actions

Key

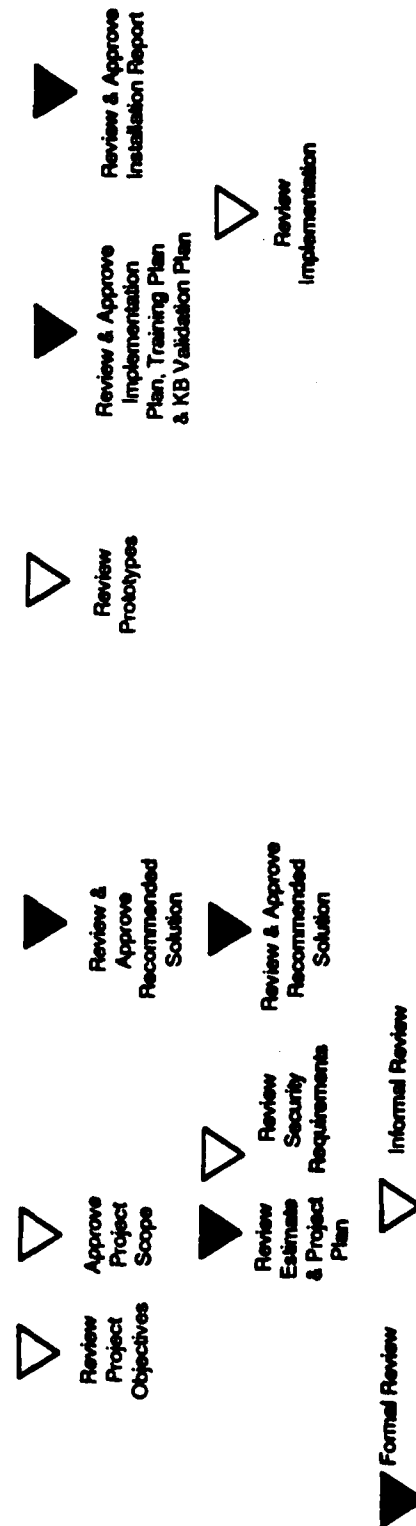


Figure 3-4 Management and Quality Assurance Actions

review of project objectives, approval of project scope, possibly also a project budget at this point. During the prototyping phase management should also informally review each prototype to make sure it conforms to the objectives of the project. Formal reviews of the project include a review and approval of the recommended solution which is generally in the form of a requirements definition or solution definition document. To review and approve the implementation plan the training plan and knowledge based validation plan and finally to review and approve the final installation report.

Quality assurance actions are also important throughout the project. Informal quality assurance review is required for security requirements on the system, and to review the implementation, in particular to determine whether or not the users are satisfied with the system. Formal reviews for quality assurance include a review of the project estimates and the project plan to review and approve the recommended solution along with management of the organization. The combination of management and quality assurance actions insures a smoothly running project.

3.6 Route Maps.

Creating a route map is the process of defining a sequence of events which will lead to the completion of an expert system implementation. The fundamental premise of the iterative development methodology is there are an unlimited number of variations of the work plan. The construction of the route map specifies each stage to be conducted and the number of iterations it is to be repeated.

Figure 3-5 shows the steps in constructing a route map through the methodology. The chapter notations beside each box is the chapter or chapters in this development methodology where the process is discussed. The organization profile in Figure 3-6 is an initial

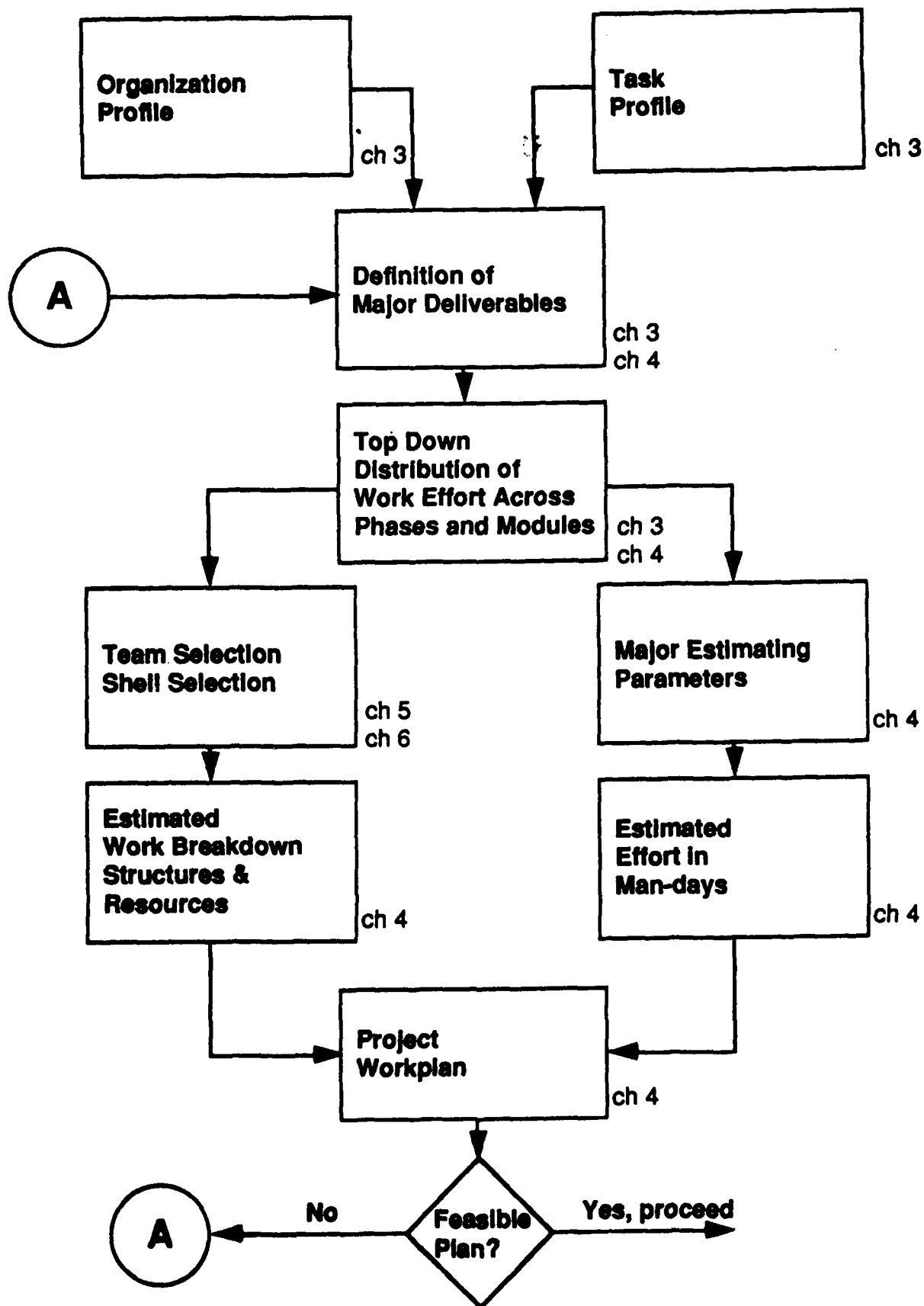


Figure 3-5. Route Map Creation

ORGANIZATION PROFILE

PROFILE CHARACTERISTICS	POSITION ON A SCALE OF 1 TO 4				RANGE OF VALUES
	1	2	3	4	
SIZE					SMALL → LARGE
COMPLEXITY					SIMPLE → COMPLEX
PENETRATION OF IT					LIMITED → EXTENSIVE
IT SOPHISTICATION					BASIC → SOPHISTICATED
ATTITUDE TO CHANGE					RESISTANT → PROACTIVE
BUSINESS PLANNING APPROACH					SHORT TERM, INFORMAL → LONG TERM, FORMAL
GENERAL CHARACTERISTICS					
LIST PRIMARY MISSIONS					

Figure 3-6 Organization Profile

step in the project initiation activates to evaluate the mission and duties of the potential sponsor organization. This becomes the needs and expectations for the system. Figure 3-7 shows a task profile to evaluate the tasks and subtasks of the organization. This will also show how the proposed system might affect the organization after implementation. Both of these are used as input to define the scope of the expert system. The scope will be used in the creation of the route map through the methodology to define a project. Careful examination of these environmental factors early in the project will prevent disappointment in the expert system later.

TASK PROFILE				
FOCUS	RELATIVE EMPHASIS ON A SCALE OF 1 (LOW) TO 4 (HIGH)			
	1	2	3	4
ORGANIZATIONAL OPPORTUNITIES				
IMPROVED MISSION EFFECTIVENESS				
IMPROVED IT EFFECTIVENESS				
ARCHITECTURES <ul style="list-style-type: none"> - Applications & Data - Technology - Management & Organisation 				
DESIGN LEADING TO SYSTEMS DEVELOPMENT				
IMPLEMENTATION				
SCOPE				
EXPECTED DURATION OR COMPLETION DATE	EXPECTED RANGE OF COSTS		KEY OBJECTIVES	

Figure 3-7 Task Profile

Chapter 4

Project Initiation and Planning

4.1 Introduction.

Project initiation begins with identification of problem types that require the style of programming underlying expert systems. Keep in mind that expert systems technology is a programming approach and therefore is neutral to the problem it can solve. But some classes of problems can be solved much easier with expert systems programming than with conventional programming. This chapter discusses the principles that can be used to identify those problems that are easier to solve with expert systems programming than with conventional programming. The chapter also provides procedures for planning a project, performing estimation of the resources required for the project, and creating a work plan for the project.

4.2 Identification of Opportunities.

Expert systems are an approach to programming that enable computers to assist humans in analyzing and solving complex problems. They extend the power of computers beyond the transactional problems and support decision making activity in semi-structured decision making environments. By encoding the knowledge of human experts, the computer can assist in diagnosing and troubleshooting failures in machinery, assisting in mobilization planning and scheduling for maintenance activities.

Equally important, some expert systems techniques can improve the human-machine interaction by making computers easier to use in such areas as database query. Another characteristic of expert systems is they make the knowledge in the system easy to revise by non-programmers. The system is easier to maintain and to understand

because the rules which represent the knowledge of the system are written in an English-like language which almost anyone can understand. Thus programming in rule based systems can become "programming for everybody."

Let's look at an example of an expert system in order to understand the problem characteristics which are important to identifying an expert system application. The application is KLUE, an expert system designed and built at 3M Corporation in Minneapolis. The system's objective is to transfer knowledge from process engineers to process operators at 3M. Process engineers at 3M define specifications for processes that ensure acceptable product quality. Various measuring devices are also used to measure parameters to determine if the processes are in control. When operators notice a significant difference in parameter readings, they must call a process engineer to diagnose the problem.

The expert system is designed to replace the need to call the engineer for most problems. When operators notice variation in parameters, they can go to the expert system and enter the current readings. The expert system then will help them diagnose the problem without the need to call an engineer. The system is now in use in process diagnosis, technical service and training. The number of users is continuing to grow.

4.2.1 Motivation to Build Expert Systems. What was the motivation to build KLUE? 3M recognized that process engineers were a scarce knowledge resource. The time lag of getting an engineer to the malfunctioning process was costly because of the delays in production. The motivation to build KLUE relied on business benefits that extended the knowledge of experts to "novices" thus enabling the novices to perform a task once performed by engineers. This is the first problem area that expert systems can address: there is a shortage of knowledge in a narrow domain of expertise.

Implementation of this knowledge often requires some judgment to be exercised in its use.

KLUE supports decisions that must be made by several individuals many times. These decisions have a measure of repetition associated with them that is both cross-sectional as well as temporal. This characteristic of having to make a relatively large number of decisions simply means that the knowledge to make the decisions will be scarce and the need to make the decisions will be high. It makes no sense to spend the resources building systems to support a single decision maker who makes a decision once or very few times. The repetition of the decision is not high enough to provide a "return on the investment in the system."

The KLUE system expanded the knowledge capabilities of the process operator. The process operators had a process engineer at their disposal to ask hypothetical questions. The result was that the system served as an excellent training vehicle both for new process engineers and process operators. Any area of the organization where on-the-job-training is important is a potential expert system. In addition, technical support and maintenance personnel began to use KLUE. Thus, the knowledge of the process engineers was distributed around the organization.

KLUE also provides the process operator with another view of his tasks. Organizations divide labor into categories that sometimes cause individuals to function within their own small "world." These subdivisions in the organization can cause conflicts in organizational goals. The expert system can help break down these barriers by giving operators another "view of the world." Thus in situations where various units of an organization must coordinate, the expert system can be helpful.

Expert systems are viable only when experts can be identified in a relatively narrow domain of expertise. The motivation for KLUE was founded in the principle that engineers knew more than operators and more engineers were needed to perform their tasks. The difference between the "novice" and the "expert" was transparent to all who looked at the problem. The expertise was the type that could be codified into rules that look like the following:

IF (washer warning light is red)
and (water dribbles)
and (pressure gauge reading less than 40 psi)
THEN (pump malfunction).

Thus, problems that can be put into expert systems have the following attributes: several experts that can be identified, all can agree that they are experts and their knowledge can be codified into rules. The task supported is also a mental task as opposed to a physical task. Since expert systems are software, they typically support mental activities.⁵

4.2.2 The Role of the System. The roles an expert system will play in the organization after it has been implemented into the organization are important to determining if an expert system is right for an organization. We must determine the future and the current missions of the organization, then look at the proposed task and determine the value of this task to both present and future operations.

The expert system usually operates in an expert role supporting users of the recommended actions from the system. KLUE for example served both as a communication and translation device between

⁵ Although one might argue that robotics is an exception to this characteristic, the expert system part of a robot typically is handling mental tasks that support the robot, and not any physical tasks.

different task domains: the engineer and the operator. The system might also support human reasoning, or perform limited reasoning. For example, KLUE can be used to determine the cause of problems with a process from parameter values.

The system might also perform knowledge accounting by maintaining a history of the decision making process. For example, KLUE stores each session into a file that can be looked at later by engineers. The engineers can look for repetitive problems that might be cause for revising an entire procedure. Engineers are freed to do this type of analytical work because they are not responding to trouble shooting calls the system is handling. The accumulated knowledge can thus be used to improve the system.

Another role of the system might be knowledge synthesis, gathering knowledge together for another expert system or human to analyze. KLUE does not do this but another system ACE, designed by AT&T, pulls together diverse data from numerous switches and presents them in a coherent manner to a maintenance technician. The system can detect switches that may fail soon and therefore the system presents alarms to the technicians. This type of synthesis prevents a technician from having to read all the raw data in order to detect impending switch failure.

4.3 Estimating the Project and Preparing the Workplan.

Estimating the project requires the preparation of a project scope. The scope of the project will determine the size of the system. The scope will also determine the level of resource effort required to accomplish the project. Estimating the level of effort for any computer system development is a difficult problem at best, impossible at worst. There is very little data with which to build models on the development of expert systems; this makes estimation of expert systems extremely difficult. The results for estimating projects given in this chapter are from the experience gained in

building 10 expert systems that have been built at Coopers & Lybrand, one of the world's largest public accounting firms.

4.3.1 Estimating the Number of Rules, Frames and Objects. The most difficult stage in the estimation process is to estimate size of the final expert system in terms of the number of rules, frames and objects. This estimate is key because this single number will be used to create the reminder of the estimate and work plan. There is no easy way to do this; but the following procedures are offered: (1) have single expert estimate the number subjectively, (2) convene a panel of experts to estimate the number, and (3) perform a pilot project to get experience then use the expert's estimate. Any estimate, no matter how much research is done, will ultimately be only a forecast subject to the subjective biases of all forecasts. But we must start somewhere.

4.3.1.1 Single Subjective Estimate. In making an estimate using a single domain expert, try to choose the best expert available. Have the expert make three or four estimates over a period of a week, with each estimate separated by two or three days. Take the average of the estimates given by the expert as the size of the system.

4.3.1.2 Panel of Experts. Enlist a panel of several experts of up to 10. Give each expert a copy of the project description prepared in the project initiation phase (see Chapter 11 for project description) so that each expert has an idea of the scope of the uses and objectives of the system. Have each expert make an independent estimate of the size of the system. Collect all the estimates and summarize into a table without identifying which expert made each estimate. Ask each expert to reconsider his estimate in light of the collective judgment of the panel. Repeat the process of collecting the second estimate and returning the second round estimates to the experts and ask for a third estimate.

After the third round average the estimates to get the size estimate for use in projecting the scope of the project.

4.3.1.3 Pilot Projects. A pilot project is a small development of a candidate expert system. The pilot project can be a microcosm of a full development of an expert system. Pilot projects generally consist of 2 to 5 weeks of effort by an expert or knowledge engineer or both to build a simplistic version of the projected system. The results of the pilot are then used to estimate the size of the project, costs, and technical risk of a full development. Pilot projects are primarily done on projects that are expected to be rather large and where more accurate data are needed in the estimation process. A pilot project usually also includes a cost benefit study as a portion of the pilot. (See Volume 2.)

4.3.2 Estimating the Effort for Project. This resulting estimate of the number of frames, rules and objects from the above procedure is the number which is utilized to enter Table 4-1 to determine the effort and schedule for that system. This table has been prepared based on 10 expert systems that have been developed by Coopers & Lybrand.⁶

These estimation guidelines assume that a trained knowledge engineering team is available to perform all the work and the problem is of medium difficulty. Therefore, the times in Table 4-1 are probably fairly close to minimum times that would be required. The table also assumes that we are working on a project of medium difficulty. Projects of extreme difficulty would take longer, projects that are fairly simple might take a little bit less. The basic assumptions in Table 4-1 are an experienced team

⁶ Remember any estimate from this table is still subjective even though it may appear to be very accurate.

Table 4-1
Estimated Effort for Expert System Development

Rules	KRules	MM	TDEV	FSP	EX
200	0.2	4.8	4.8	1.0	1.4
300	0.3	7.0	5.8	1.2	2.1
500	0.5	11.3	7.4	1.5	3.4
800	0.8	17.4	9.3	1.9	5.2
1000	1.0	21.5	10.4	2.1	6.5
1500	1.5	31.4	12.6	2.5	9.4
2000	2.0	41.0	14.5	2.8	12.3
2500	2.5	50.5	16.1	3.1	15.2
3000	3.0	59.9	17.6	3.4	18.0
3500	3.5	69.2	19.0	3.6	20.8
5000	5.0	96.5	22.6	4.3	29.0
12000	12.0	218.4	34.6	6.3	65.5
20000	20.0	352.8	44.3	7.9	105.3

Key: Rules = number of rules frames and objects
 KRules = 1000 rules; MM = man-months; TDEV =
 calendar months for development; FSP = full time
 software personnel; EX = man-months of expert time.

performing the work, and a medium difficulty problem as perceived by the development team.

4.3.3 Preparing the Workplan. Determining the work breakdown structure and the distribution of the effort of the project is the next step in making the project time and materials estimate. The task for creating an expert system has been broken down into six basic steps: Requirements definition, solution definition, knowledge acquisition, design, prototyping, and test/transition. These steps are shown in Table 4-2 along with the allocation of the resource effort required in each phase. Table 4-2 shows both effort distribution, that is how the man months (MM) should be distributed, as well as the schedule or time of development (TDEV) distribution. Both of these charts are based on the software

personnel man months and the development time for the project from Table 4-1.

We usually call systems with 800 rules or less a small system. Similarly, intermediate systems are 1000 to 3000 rules, medium are 3000 to 8000 rules and large are 8000 and larger. Use these designations and the actual values obtained from Table 4-1 to estimate the distribution from Table 4-2.

Table 4-2
Profile of Effort and Schedule by Product Size

Phase	System Size			
	Small %	Interm %	Medium %	Large %
Effort (MM)				
Requirements Definition	6	6	6	6
Solution Definition	7	7	7	7
Knowledge Acquisition	20	20	20	20
Design	7	7	7	7
Prototyping	45	42	39	37
Test/Transition	15	18	21	23
Totals	100	100	100	100
Schedule (TDEV)				
Requirements Definition	8	8	8	8
Solution Definition	8	8	8	8
Knowledge Acquisition	20	18	16	14
Design	7	7	7	7
Prototyping	40	38	36	34
Test/Transition	17	21	25	29
Totals	100	100	100	100

As can be seen in the table for intermediate sized systems, requirements definition requires 6% of the effort that is expended but will utilize 8% of the development time. Solution definition

will require 7% of the effort but 8% of the schedule time. Knowledge acquisition requires 20% of the development effort but only 18% of the schedule time. The design process requires 7% of development effort and also 7% of schedule time. Prototyping phase requires 42% of the effort, but 38% of the schedule. The test/transition phases require approximately 18% of the development effort, but 21% of the schedule time.

4.3.4 Putting it all together in an example. As an example of how we might estimate a project, let's assume that we have estimated this project will be an expert system that will have 1,500 rules in the system. Based upon this initial estimate of rules we can enter Table 4-1 and determine that this will require 31.4 man-months of full time software personnel and 9.4 man-months of domain expert time. In addition, schedule time to complete the project should be 12.6 months.

Using the guidelines for Table 4-2, we conclude that a 1,500 rule system is an intermediate size. We enter Table 4-2 with intermediate to obtain the following work breakdown structure for the effort in programming:

Requirements Definition	6%	1.9 MM
Solution Definition	7%	2.2 MM
Knowledge Acquisition	20%	6.3 MM
Design	7%	2.2 MM
Prototyping	45%	13.2 MM
Test/transition	18%	5.6 MM

In addition to the approximately 2.5 full time software personnel required on the project, about 9.4 MM of domain expert time will be required. This can be one or several experts. The time must be split between the knowledge acquisition sessions and the evaluation of the prototypes.

Table 4-1 shows this project should take 14.5 months of calendar time to complete. The work breakdown structure is not performed sequentially, however. Rather, it is in iterative cycles. We try to deliver a prototype about every 60 - 90 days for evaluation by the expert or user community or both. Using this guideline, an approximate one year project will have 4 or 5 prototypes. For this project we decide to have four prototype deliveries on the project and divide the knowledge acquisition, design and prototyping into four equal parts.

Figure 4-1 shows a Gantt chart that provides the schedule for the activities of this example project to build an expert system. Notice how the various tasks have been incorporated into a project schedule that includes each subtask. Some are sequential and some can be done in parallel. The prototyping phase overlaps design as well as knowledge acquisition. The test/transition phase also overlaps the prototyping phase by 1/2 month or so. These overlap periods provide times when the team interacts very heavily to get the prototype running properly. The last two months of the project are spent in testing and validation. Testing and validation of the system, particularly knowledge based validation will be discussed more thoroughly in Volume 2, Evaluation Methodology.

This example provides a straight forward analysis of what an intermediate sized system might look like in estimating the project and defining work breakdown structure and schedule.

Plans &
Requirements

Knowledge &
Acquisition

Design

Prototyping

Test &
Transition

0 1 2 3 4 5 6 7 8 9 10 11 12 16

Time Months

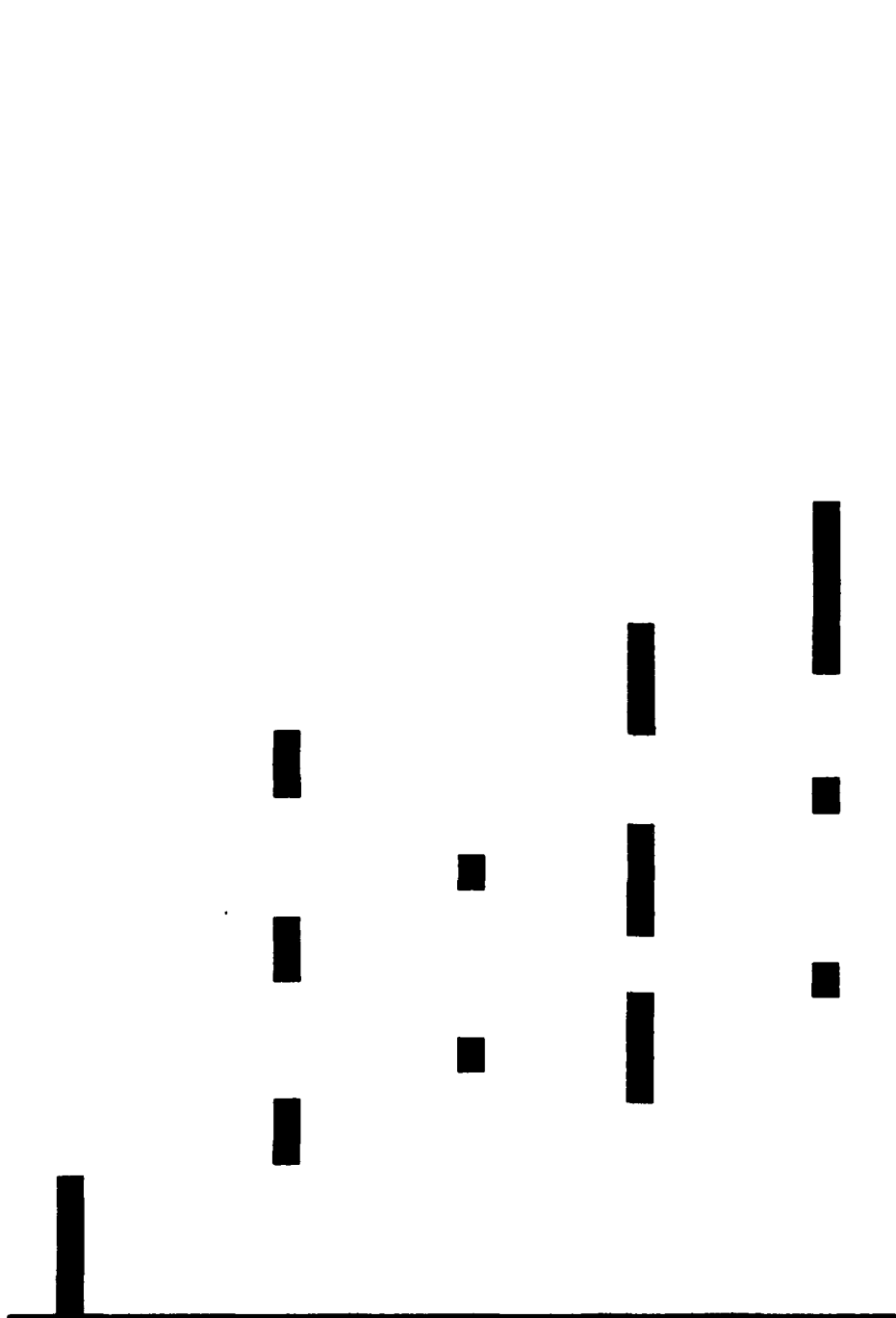


Figure 4-1 Gantt Chart of 1500 Rule System

Chapter 5

Selection of the Development Team

5.1 Introduction.

The makeup of the development team is critical to the success of the project and to the long term viability of the expert system produced. Team members should be selected for their job related skills. Expert system shells for building the system need to be selected based on a set of multiple criteria to get the best expert system shell for the current project. This chapter will discuss the requirements that are necessary in selecting the individuals to serve on a team: knowledge engineer, human experts, and users.

The term knowledge engineer has been coined to describe the professionals who carry out the expert system development process. He/she is the "lead builder" of the system. In a more conventional development, he/she might be called "systems analyst" or "analyst/programmer." When the expert systems development is compared to a more traditional development, many differences are found which suggest the knowledge engineer must possess competencies not previously required. A broader range of skills is required and these skills go beyond technical programming or analyst knowledge in the computer science field.

The domain expert is the individual(s) on the team possessing the expertise of the domain to be emulated. She/he is the knowledge source. The domain expert is obviously the key individual on the team, since without him/her there will be no system.

The user is a representative of the using community. She/he will provide the user perspective to the development team.

5.2 Selection of the Team.

5.2.1 Appointment of the Team. The team members should be appointed from two broad groups. First, knowledge engineer(s) who is a specialist in the area of development of expert systems. Second, there is a stakeholder group. This group may consist of various members of the organization developing the expert system, such as the domain expert community, middle management, top management, and data processing department. These stakeholders may view the project from the perspectives of users, decision makers, knowledge sources, project champions, and skeptics (devil's advocates).

Successful projects depend on the melding of all viewpoints of the organization into the common goal of implementation of the expert system. Various studies have shown the single most important element in organizational acceptance of computer systems is stakeholder involvement in the development of the system from the beginning of the project. This has led to an extremely important assumption in the selection of the team known as the **stakeholder assumption**.

Stakeholder Assumption - key people who have a stake in the success of the project should be actively and meaningfully involved in shaping the project in order to focus the project on meaningful and appropriate issues, thereby increasing the likelihood of successful implementation.

The team is identified and appointed which includes knowledge engineers, domain experts, and users. The team is a vehicle for actively involving the key stakeholders in the project. This team is organized to make major decisions about the focus, methods, hardware/software issues, goals, installations, and testing/evaluation of the project. The team shares the responsibility for decision making about system development with the knowledge engineer by providing a forum for the perspectives

of the stakeholders who will ultimately be involved with the implementation of the system.

At least four benefits in the project development accrue to the project by having users involved in the development.

- * The domain experts and users understand the goals and objectives of the expert system as well as the limitations of the system.
- * There is much better communication among the knowledge engineers, users and domain experts, thereby increasing the quality of the system.
- * The users and domain experts share the responsibility for the system with the knowledge engineer and therefore "care" for the system.
- * The personal commitment of the users and domain experts substantially increases, making it much more likely that they will use the expert system.

5.2.2 Team Composition. It is important to form the right group for the team. The team will become the core group of the project. The size of this team would normally be expected to be a minimum of three (knowledge engineer, domain expert and user) and a maximum of about ten. Occasionally, a team could be two (knowledge engineer and domain expert) when the expert is also the user. It is not a good practice to have a team of one except under the condition that the developer is also the only user. The reason the expert cannot be his/her own knowledge engineer will be discussed below. Of course, the core team may be expanded as necessary to provide the appropriate level of effort by including more experts, users, programmers, etc.

5.2.3 Selection of Knowledge Engineers. Selection of knowledge engineers is relatively simple when experienced expert systems builders are available. The assumption of this section, however,

is that you are selecting individuals for the team that do not have specific experience in knowledge engineering. The objectives for your selection can be long term, e.g. to build an expert systems group, or short term for a single or few projects.

The knowledge engineer must have skills in the following areas: interpersonal skills, analytical and problem solving ability, an adaptive work style, some experience in the domain, and high motivation. First, the knowledge engineer must have excellent interpersonal skills so that he/she can lead and manage the team as well as look after the requirements desired by the user. This individual must understand how to elicit expert knowledge and what the users of the expert system will want. The knowledge engineer must have the ability to do interviewing so that he/she can interview experts and users.

The knowledge engineer must be able to use his/her communication skills to gain the confidence of the domain experts on the project. This is not always easy. Many experts are initially concerned about losing their jobs to computers and others have little understanding about how computers work. The knowledge engineer must therefore be willing and able to teach the expert about expert systems to make him more useful to the project.

The knowledge engineer should be comfortable with at least one computer language and such facets of computer programming such as graphics interfaces, data base access, and nonstandard input and output. The knowledge engineer should be a skilled analyst with experience in breaking problems into their components, evaluating alternatives, and managing phased implementations.

The knowledge engineer must possess analytical skills and knowledge modeling skills. They must have knowledge of expert system architecture as well as the direction the application will be

taking. The knowledge engineer must have conceptual synthesis capabilities to be able to take information elicited from the domain expert, and then translate that into generic problem solving models. These models can be captured into knowledge diagrams or directly into the expert system rules.

She/he must also have a degree of enthusiasm for and knowledge of the subject matter that the domain will be for the expert system. It's extremely important that knowledge engineers working on the system have some knowledge about the problem domain that will be the basis for the application. The knowledge engineer should not and need not be an expert in this area. If he is an expert in the domain area there will be a tendency for the knowledge engineer to superimpose his knowledge on top of the domain experts. This is not good. However, if the knowledge engineer has no knowledge of the domain the knowledge acquisition problem will be significantly more difficult because the knowledge engineer will be unfamiliar with vocabulary and various other aspects of the domain. It may also cause conflict between the knowledge engineer and the expert. Experts are individuals who tend not to like to have their time wasted. When a knowledge engineer exhibits no knowledge whatsoever of the expert's domain of expertise, the expert will tend not to have respect for that knowledge engineer.

The knowledge engineer is the caretaker of the team and is the leader of the project development. She/he must have the usual skills of management and leadership and be able to balance various personalities and egos on the team.

5.2.4 Selection of stakeholders. The following general selection criteria may be used in choosing user and domain expert members of the team. Not all of these criteria can be met to the same degree in every case, but these form a basic set for considering stakeholder members.

- * They represent interested groups, constituencies and expertise.
- * They have the authority to use the system in decision making situations.
- * They have a fundamental belief in the technology and value of the expert system.
- * They are willing to make the commitment of time required to "do it right."

5.2.4.1 Domain Experts. Experts that work in the application team must also be carefully chosen. Some experts are not comfortable with the design process. In addition, some experts have difficulty with the type of introspective analysis required to accurately communicate their underlying knowledge to others. Domain experts should be chosen with the following characteristics in mind:

- * A high level of competence in the problem to be solved.
- * The ability to communicate internal problem-solving process well.
- * A flexible and inquiring intellectual style.
- * Interest in the expert system building process and willingness persist in the development.
- * Absence of felt threat from the expert system.
- * Interpersonal style amenable to interactively working with team members.

It is imperative that the experts selected to operate on the development team have the capability for expressing themselves well and being able to communicate their thought processes in their area of expertise. These thought processes will have to be prompted from the knowledge engineer through various knowledge acquisition techniques.

The domain expert should also have some enthusiasm for working on the project. Domain experts that are hostile to the project will

tend to create a substandard expert system. However, it should also be recognized that most experts will be hostile to an expert system development project in the beginning. What must be evaluated is how deep this hostility is and whether or not it can be overcome through good knowledge engineering techniques.

5.2.4.2 Difficulty of Knowledge Engineering. The most difficult part of knowledge engineering is realizing that experts often do not know how they make decisions. The real decision making processes used by an expert is much different from formal knowledge. Many experts cannot discriminate between the two. It is up to the knowledge engineer to discover the unique rules of thumb used by the expert by cutting the formal outer layers of knowledge.

Experts will have extreme difficulty performing their own knowledge engineering. There are certain psychological blind spots that exist in every human's mind, domain experts included. When experts attempt to build their own system, they often build a system according to what they think they do. In fact, many experts perform tasks differently than what they have cognitively allowed themselves to believe. In other words, experts often perform many portions of their task inside their domain of expertise in a very automatic way. And, they do not, themselves, understand that they are performing certain components of the task so automatically.

As an example, consider driving down the expressway and noticing a billboard alongside that expressway that perhaps advertises a product that you have a need for. As you look at that billboard you might understand that this billboard has been in that same location for months, perhaps even years, and yet today as you pass it, you see it for the first time. Why is that true? Because the mind contains a filtering mechanism which filters out most of the

stimuli that come into our brain each day. If this filter mechanism did not work, we would have no attention span whatsoever. It would be impossible to do anything because every stimulus that came into our brain would distract us.

The brain has developed methods of filtering certain stimuli in order to allow us to concentrate. We notice certain stimuli only when we need them. This is why we can drive down the road past a particular billboard for days, weeks, months, and years and never see that billboard nor know that it's there until we have a need for that product. Suddenly it pops out at us.

As another example, suppose you're going home from work. This is an automatic process because you do it every day. You walk from the office, get in the car, begin driving, generally speaking you will have a particular route that you take home every single day. And then as you're driving home one day you're about half way there it suddenly pops into your mind that you should have gone by the cleaners on the way home to pick up the cleaning.

Why did you forget the laundry? The reason is, again, that you went into an automatic path in your mind. Your mind had certain things that it did at a particular time of day and it does it that way every day. This road map of decision processes of activities that you perform in order to get home becomes an automatic, everyday occurrence. You cannot change it without very willful and conscious processes to make that change. In other words, if you want to remember to pick up the laundry on the way home, you write yourself a note and stick it in an obvious place so that you will see it to remind you.

The brain does this because it can perform pattern matching activities in long term memory much better than in short term memory. The mind stores items in long term memory with a process

that psychologists call compiling. It is very similar to compiling a computer program. Long term memory is maintained in the brain in this compiled format. This is where pattern matching activities take place most efficiently. It is here that automatic operations take place. We never have to call them into short memory.

In order to retrieve long term memory or to "un-compile" these long term memory patterns, an external stimulus will often be required. These external stimuli usually require an evaluating mechanism to assist the domain expert in articulating his knowledge. While it is possible for some experts to do this alone, it is usually the case that the process is much more efficient with knowledge engineer assistance.

The point to be made here is that the domain expert will find being his/her own knowledge engineer exceedingly difficult. The worst part of it is that they will usually not even realize the deficiencies in the system until someone else tries to use it. If the expert is using his own system, he might be able to act as his own knowledge engineer, especially if it's a small system, say 300 rules or less. However, as systems get larger, external interaction with the domain expert will be required in order to build the system that correctly defines the methods used by the domain expert.

The team should appoint a chairman to direct the group. One of the team participants should be designated as chair rather than a knowledge engineer. Having a domain expert or decision maker as chair of the group helps symbolize the responsibility and authority of the team. The knowledge engineer must be the collaborator, facilitator, and trainer. He commands a good deal of floor time; however, the knowledge engineer can accomplish everything needed in the project by working through the chair rather than by being the chair.

Chapter 6

Selection of the Expert System Shell

6.1 Introduction

The expert system shell (or shell) is the development tool or language in which the system will be written. A shell provides development team support through the development interface and user interaction through the user interface. The shell will define the knowledge representation scheme used by the system, reasoning method (forward or backward chaining), development platform, delivery platform, and compilation capabilities for runtime distributions. Remember, almost any system can be written in almost any language; the key is to try to minimize the development effort and time with an appropriate expert system shell.

Selection of the expert system shell for the development project is based on a set of multiple criteria: problem type, user interface, development interface, method of reasoning, development platform, delivery platform, compilation capabilities, and vendor support functions. Each of these attributes has a requirements side and a capabilities side. That is, we have user interface requirements to solve our problem, and each shell will have user interface capabilities. As we look at each attribute, consider first your problem requirements, then evaluate the shell on its capability to meet your requirements. Usually, a single attribute will cause overriding constraints on the shell selection, e.g. delivery platform, which limit the search to a subset of all available shells.

We will discuss each of these criteria in turn in this chapter. Observe that the objective of this section is not to select "the best shell currently on the market" but rather to give you the

criteria with which to make the selection. Any survey of existing shells would be incomplete and inevitably out of date since there are hundreds of shells on the market today and more coming all the time. By providing general principles, this guide will enable you to make an intelligent and rational choice on any shell.

6.2 Shell Selection Method.

The task of selecting a shell for the development team is balancing the domain environment with the capabilities of shells at an acceptable cost. In order to make an intelligent choice of a shell, there are two essential classes of information to gather: requirements of the problem environment and capabilities of the candidate shells. To select a shell, you must:

- * categorize your problem type,
- * define interface requirements,
- * define platform requirements,
- * determine shell characteristics of the shells of interest and
- * select the shell which best matches its capabilities to the requirements.

6.3 Categorize your problem type.

The first step in selecting a shell is categorizing your problem into a standard problem type category. Not all shells are appropriate for solving all types of problems. The following descriptions of problem types will enable you to categorize your problem into one of the classes. Although almost any problem can be solved with almost any shell, it is true that choosing a shell that is particularly appropriate for your problem type will make the development easier and faster.

A good deal of confusion exists in the selection of shells because problem solving strategies and programming techniques in the shells have been confused with application problem types. Other confusion

exists because not every problem can be solved with every problem solving strategy. Therefore, it's well to learn a little bit about problem types as well as problem solving strategies and programming techniques that can be utilized in an expert system shell.

6.3.1 Diagnostic Problems. Diagnostic problems refer to problems that involve making recommendations after asking the user a number of questions. Diagnostic problems are characterized by situations which are found malfunctioning in some way. To determine the cause of the malfunction, the diagnostician must ask for an amount of data, then based on the data describing the system state, an action is required to put the situation back to normal. A physician practices diagnosis when she asks questions of a patient with a health complaint in the process of trying to determine which of several disease organisms may be present. By the same token, an auto mechanic practices diagnosis when he asks the car's owner questions and conducts tests to determine what parts of the automobile is causing a problem and how it should be fixed.

Frequently diagnostic systems will incorporate specialized techniques to make the application more useful to a particular industry. For example, many financial systems combine diagnosis with elements of planning and scheduling to create an expert system for recommending portfolios.

6.3.2 Monitoring Systems. Monitoring problems begin with input and reach one of a limited sets of recommendations. In a sense, they are similar to diagnostic problems except diagnostic problems are typically reached via backward chaining, while monitoring problems are more often handled with forward chaining systems. A typical monitoring problem might be watching a typical data stream and alerting the user when an unusual pattern occurs. Thus a monitoring system might collect data on telecommunications switches and notify an engineer when test parameters indicate trouble at a

specific location in the network.

Process control systems in manufacturing are typically monitoring problems. These applications are frequently subdivided into open systems and closed loop systems. Open loop systems warn the operator that he or she should consider an action recommended by the expert system. Closed loop systems are where the expert system analyses the problem and initiates the appropriate action on its own.

6.3.3 Design Problems. Design problems involve selecting an appropriate way to combine components. Configuration systems that assemble a complete computer system from an initial set of customer specifications is one example of a design system. Design problems begin with inputs and then determine the constraints implied by the inputs. A solution is then found that violates no constraints and meets the objective of the initial problem. For example, if the user requests a cooling fan with 3-inch blades that produces an airflow of 5 cubic feet per minute, then all other designs are automatically excluded from consideration. Conflict resolution is the key programming strategy to a sophisticated design system.

6.3.4 Scheduling Problems. Scheduling problems involves ordering a set of components and actions within a time sequence. These problems typically involve discrete manufacturing scheduling (flow shop scheduling) and on-demand scheduling (job shop scheduling) applications. In all cases, the systems coordinate people, machinery, and job tasks that must be accomplished within some time frame.

For example, MOBPLEX has a scheduling module within it that schedules the ranges, ammunition and transportation assets required to bring a unit up to the proper readiness category for shipment to the theater of operations. Scheduling systems require

constraint propagation facilities, conflict resolution truth maintenance, and hypothetical reasoning.

6.3.5 Planning Problems. Planning problems are sometimes treated as a subset of design problems and sometimes classified as scheduling problems. If a problem involves specifying a set of components that will be necessary for a plan (e.g., personnel, resources, budget) then it may be very similar to the design problems we have just described. If it involves developing a schedule or interactively guiding the execution of a plan, it should probably be considered a scheduling problem.

6.4 Define Interface Requirements.

6.4.1 User interface. The end user interface is another important characteristic to look for when selecting a shell. Generally speaking the smaller and simpler shells that run on a PC, particularly those that do backward chaining, will only allow question and answer dialogue as a user interface. These systems require the user to enter a yes/no or multiple choice response to the questions. This may be appropriate for many classes of problems, particularly diagnostic problems, but it may not be appropriate for an insurance underwriting system which is based upon many different inputs. In this case, the user interface might more appropriately be a business forms approach. At the same time the output to the user might well be a graphic rather than simply directions to perform some action.

Another feature of the end user interface that's very important is whether or not the system provides for on-line help. Given the nature of most expert systems' tasks it is highly desirable to have on-line help features available inside the shell. Expert systems as a technology imply the capability for explanation, namely to explain to the user WHY a particular piece of information is necessary and to explain HOW certain conclusions of the expert

system were reached. These explanation facilities to the user should be present, and depending upon the kind of system you're building, these features will vary.

Another end user important item is the response time of the interface. These user interface considerations may well determine how well the user accepts the system. So we must give consideration to the amount of time the user spends interacting with the system. The time interacting with the system is both a design consideration as well as a function of the delivery platform and software combination.

6.4.2 Development Interface. The development interface is another characteristic that is very important in the selection of a shell. The knowledge based creation activity may be relatively difficult or simple depending upon the kind of shell. When you're dealing with small and simple type shells, generally speaking knowledge based creation will be done either on a word processor of your choice or a word processor that comes with a shell. These tend to be basically rule based systems. However, as you move into more graphic oriented development such as an object oriented system, the ability to create graphics in the development environment will become extremely important.

Most of the large complex shells have the capability for graphics user interfaces that can be easily designed and implemented by the developer. This graphic support may be extremely important in certain diagnostic systems for example, or simulation systems or training systems that may be implemented. During knowledge based development and during system development the developer will often have to do debugging of the system. The capability for graphic knowledge based displays and inference tracing inside of the system will be very important to efficient programming of the system.

The large hybrid systems (see paragraph 6.6) tend to have very good inference tracing and graphic knowledge base displays. Small systems particularly those that run on the PC tend to have very little support in the area of debugging and particularly in graphic knowledge based displays in inference tracing. Another important aspect of the development environment is the capability to insert a WHY and HOW explanations directly by the developer rather than having the system generate them. Some systems generate WHY explanations for example, simply by showing the rules on the screen the system is attempting to satisfy. This rule may not be in a form and format which are easily interpreted by the user. Therefore, it is very important the developer have the capability to put in his own WHY and HOW explanations in a form the user will be able to understand.

6.4.3 System Interfaces. The next characteristic that's important are the system interfaces that are available to the developer inside the shell. If the system is going to be a stand alone system running only on a PC, then the system integration is not as much of an issue; but, if the system must integrate with other existing data bases on a mainframe, then grave consideration must be given to the system interface capability of the shell. Shells have various levels of integration capabilities. Often, integration in the shell literature means that you can transfer an ASCII file into the shell, or you can transfer a data base file into the shell. Of course that data base must be put into a format that will be acceptable to the shell. More often than not a program interfacing the data base to the shell will have to be written in order to make the interface.

Hooks into other languages are also important. If you have reason to believe that you will need hooks into other languages, then you need to identify whether or not the shell has the hooks you require. Additionally, you must look into the operating system

compatibility of the shell to make sure that it runs in your operating system.

6.5 Define Platform Requirements.

You must also consider the hardware and software requirements both for the development environment and for the delivery environment. The development or delivery environment may well be an overriding consideration in selecting a shell for your project. For example, if you are constrained to a development machine of IBM-PC AT or a clone, then this hardware limitation will restrict your shell choices to small PC shells (see paragraph 6.6.)

6.5.1 Development Platform Requirements. The development platform will often be determined by the software selected. You must define requirements for your development environment in order to provide guidance on the types of shells that can be considered. Shells can be generically classified by the development platform they run on (see paragraph 6.6.) Some "PC shells", for example GoldWorks, require extensive modification to standard PC hardware in order to be used with the expert system shell. When considering development platforms, though, remember that some large shells have the capability to compile a "runtime" version that can be delivered in much smaller PC's than are required in the development environment. A word of caution though, these "runtime" may be relatively expensive if you have to distribute many copies of the expert system. The guide here is caveat emptor: buyer beware. Probe your vendor extensively on this issue.

6.5.2 Delivery Platform Requirements. The delivery of your expert system must be considered in the early planning stages of your project. If you're building a system that will be distributed widely then it's extremely important that your shell run on your existing and compatible hardware without significant upgrade. If the development hardware is significantly different from your

"runtime" environment you must be able to compile your system from the delivery environment into the "runtime" environment. It is entirely like that your delivery environment will be the largest constraint on your selection of a shell.

6.6 Determine Shell Capabilities.

Shell capabilities are founded on their development and delivery platforms and programming strategies. Platforms can be PC's, Unix work stations, LISP machine work stations, or the mainframe. Generally, the larger the platform the more capabilities the shell will have. LISP platform shell have the most capability but have the drawback of difficult integration.

6.6.1 Classes of Shells. Today, the largest number of shells are running in the IBM-PC. The platform can also be a work station defined as a SUN, MicroVAX or other Unix based computer. The platform could also be a LISP machine work station such as the Symbolics or Texas Instruments Explorer. These platforms tend to be expensive and difficult to integrate into the ordinary MIS environment. Lastly, a shell might have a mainframe platform.

Most shells allow you to insert HOW and WHY into the system. Not all allow you to customize these explanations. The customization of HOW and WHY is a very necessary capability. Look for this characteristic when you evaluate a shell.

Figure 6-1 provides a classification of some popular shells segregated by the type of development platform they are capable of running on. The shells with bold circles are the market leaders in their class. Some shells, like Level 5 and Nexpert Object, have versions that run in PC, work station and mainframe environments. This is a very desirable characteristic if you have a system that will be running in several environments.

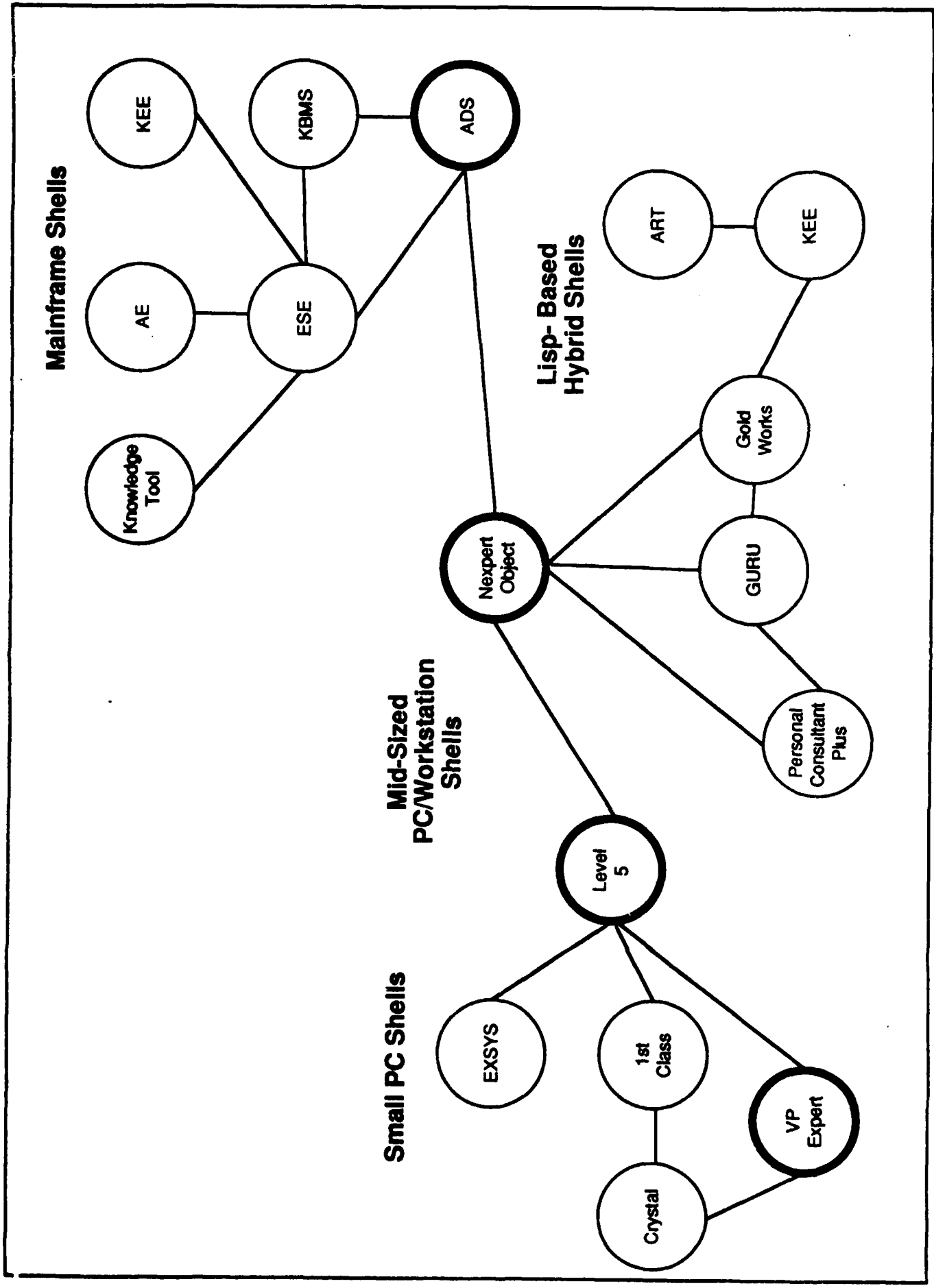


Figure 6-1 Expert System Shells

6.6.1.1. Small PC based shells typically run on ordinary PC's of the type usually found in most organizations. Their capability, generally speaking, is backward chaining rule based type techniques in their program. These are the least expensive of the shells and make an excellent place to start into expert systems.

6.6.1.2. Mid-size shells will run on work stations such as the Sun, MicroVAX, Masscomp and Apollo. These shells may have both backward chaining and forward chaining capabilities built into them. They may on occasion also have object oriented capabilities.

6.6.1.3. The large LISP based hybrid shells typically run only on special purpose LISP machines such as the Symbolics or Texas Instruments Explorer. These shells are called "hybrid" because they have both object oriented capability as well as rule based programming. These shells usually also have both forward chaining and backward chaining capabilities.

6.6.2 Shell Capabilities. Table 6-1 shows the problem solving strategies that you might find in a shell. These strategies will be discussed below. Figure 6-2 shows the relationship between the problem solving strategies and problem types above.

6.6.2.1. Abstraction refers to situations which the initial user inputs are reclassified in more abstract terms reasoned about and

Table 6-1. Problem Solving Strategies

Abstraction
Constraints and Constraint Satisfaction
Conflict Resolution
Truth Maintenance
Hypothetical Reasoning

Strategy Type	Abstraction	Constraints	Conflict Resolution	Truth Maint	Hypothetical Reasoning
Diagnostic	X	X	X		
Monitoring		X	X		
Design				X	X
Scheduling		X	X	X	X

Figure 6-2 Relationship Between Problem Solving Strategies and Problem Types

reclassified back into more concrete terms. For example, the user might be asked for specific dial reading, or a gauge value, or a number to be put in some box on some form. A rule in the system might interpret those concrete numbers to be signs of a overheated pipe or an inadequate amount of cash reserve.

6.6.2.2. Constraints and constraint satisfaction refers to situations in which the user input implies constraints on possible outputs. Thus, for example, if the user says she wants to purchase an expert system shell for under \$300, the system advisor would create an appropriate constraint mechanism that would eliminate any shells that cost more than \$300 from further consideration.

6.6.2.3 The next item, conflict resolution, refers to strategies which handle situations when a system encounters conflicting constraints. For example, if the user above who is buying an expert system shell will spend only \$300, but at the same time has requested a feature such as object oriented programming and this feature is available only on shells that cost more than \$800, then a conflict in these constraints exist which the system must be able to resolve.

6.6.2.4. The next strategy is truth maintenance. This refers to the ability of the system to update information in its working memory as new information arrives. For example, if default information suggests that all birds can fly and that George is a bird, we assume that this bird can fly. If we're later told that the object, George, is a penguin and we know that penguins are birds and that they can't fly, then we must be able to update the truth maintenance that George cannot fly, even though we know by default we would have expected him to because he's a bird. Later on we might be told that the fact that George is a penguin is a false fact. Under that condition, the system must be able to revert back to the fact that George can fly again. These

activities are generally and generically called truth maintenance inside various systems.

6.6.2.5. The next strategy is hypothetical reasoning. This is a problem solving strategy that allows the system to maintain and pursue multiple reasoning paths at the same time. Thus if we find that our user can't make up his mind on some key feature of a truck that he's designing, we can assume that he has chosen both aspects of the feature. The system will then develop two different designs, one on the assumption that he has chosen the feature, and the other on the assumption that he has not chosen the feature. Later we will likely get evidence that will rule out one of the two designs. Otherwise, the system recommends two different designs and let's the user choose which design to use.

6.6.3 Programming Techniques. In most cases, a conceptual problem solving strategy can be represented by means of different programming techniques. Programming techniques are shown in Table 6-2 below. Figure 6-3 shows the relationship between problem types and programming techniques. These include approaches to the representation of knowledge such as rules and objects. They also include backward chaining, forward chaining and certainty factors. All of the problem solving strategies presented above can be represented in rule based systems, and they can all be represented in object oriented systems as well. Generally, though rule based system shells are easier to develop and therefore tend to be the only option in the smaller (PC) shells. Object oriented programming shells require large amounts of memory to be effective, therefore, they tend to be found in large platform environments, especially work station and larger.

6.6.3.1. Rule based programming was the first type of programming of an expert system. It involves preparing structured statements of the type:

IF

(Conditions)

THEN

(Actions)

The actual syntax of the rule is dependent on the expert system shell. Rule based programming is usually the only type of programming technique found in the small PC shells. Rules are IF-THEN clauses that encapsulate knowledge as a series of antecedents that cause actions. The IF portion of the rule is called the antecedent, the THEN portion of the rule provides the action to be taken.

Table 6-2. Programming Techniques

Rules

Backward Chaining

Forward Chaining

Certainty Factors

Object Oriented Programming

6.6.3.2. Backward chaining techniques are ways of looking at rule based systems that allow us to work from a particular conclusion back toward the reasons for having that conclusion. Very often diagnostic systems are in this category. The reason being that the results are observed as a behavior of a machine or possibly dial or gauge output, or perhaps a set of indicator lights. In other words, the result is known. The reasons for these results, however, are unknown. A backward chaining system, therefore, begins with a known set of circumstances and works backward through the knowledge base to determine the causes for the results.

6.6.3.3. Forward chaining on the other hand works in exactly the opposite sense. It reaches a conclusion based upon facts that are

Strategy Type	Rules	Backward Chaining	Forward Chaining	Certainty Factors	Object or Frames
Diagnostic	X	X		X	X
Monitoring	X	X	X		
Design	X		X	X	X
Scheduling			X		X

Figure 6-3 Relationship Between Programming Technique and Problem Types

entered into the system. These facts allow the system to select rules to be activated. When a rule is activated, its conclusion, or THEN portion of the rule either provides an action for the user or another fact to activate other rules. If it is another fact, this fact will then activate other rules in the system and the process will be repeated over and over again.

6.6.3.4. Certainty factors are numeric values that allow us to perform conflict resolution (problem solving strategy.) They're particularly useful in diagnostic situations. On the other hand, certainty factors are extremely difficult to interpret for most users. In this development methodology, we recommend against the use of certainty factors. When they are used, ensure that as a developer you understand the way the certainty factors are calculated and what they truly represent.

Certainty factors range in value from -1 to +1 whereas probability numbers range from 0 to 1. Thus certainty factors are not probabilities. Be careful not to use them as probability. Certainty factors should really have been named belief factors, because what they really represent is the belief state or unbelief state of an expert as he reasons about a problem. That is, for example, a doctor might believe, based upon the evidence, that you have cancer. His belief can be high or low. His unbelief also can be high or low, that is he believes you do not have cancer. If the physician believes strongly that you have cancer, then you might say his belief factor is .9. If the physician on the other hand strongly believes that you do not have cancer as a result of the test results then you might give it a -.9.

The bottom line on certainty factors is don't use them unless you have a very sophisticated user who fully understands the implication of the factors.

6.6.3.5. The last item in Table 6-2 is objects or frames. Objects are entities that can be identified in an expert system. Objects can be placed into a hierarchy that allows general classes of objects to be defined and more specific items defined below the general object. Lower level objects are able to inherit characteristics from higher objects. Specific objects are called instances. Object hierarchies are very powerful but their drawback is the system to run them typically is large. These types of shells are usually found in the LISP machine class, work station, and mainframes. Object oriented programming is a programming technique that is used effectively in classification systems and in hypothetical reasoning areas. Objects are also very important to the design of a system because they give us the opportunity to inherit characteristics very easily, from one object to another.

Figure 6-4 provides an example of an object hierarchy. As shown, mammal is the general class, dog is a sub-class or specific group and Buzzy is an instance of dog. The characteristics of any mammal do not have to be explicitly defined for Buzzy, since he inherits them from the class object through the sub-class.

6.6.4 Vendor support and Cost. The last dimension you need to utilize in selecting a shell is the vendor support. This includes training support, documentation and the cost of the shell. In the area of training, is it necessary for you to be specifically trained on the shell, or can you train yourself? Large hybrid shells are extremely complex and typically require a training class for you to become proficient in programming it. The smaller shells, typically running in a PC, can usually be learned sufficiently adequately by simply starting to use them, just like any other PC software. Does the vendor of the software provide electronic mail, telephone hotline service, and/or consulting time to assist you when you have problems? You will almost always need some kind of support.

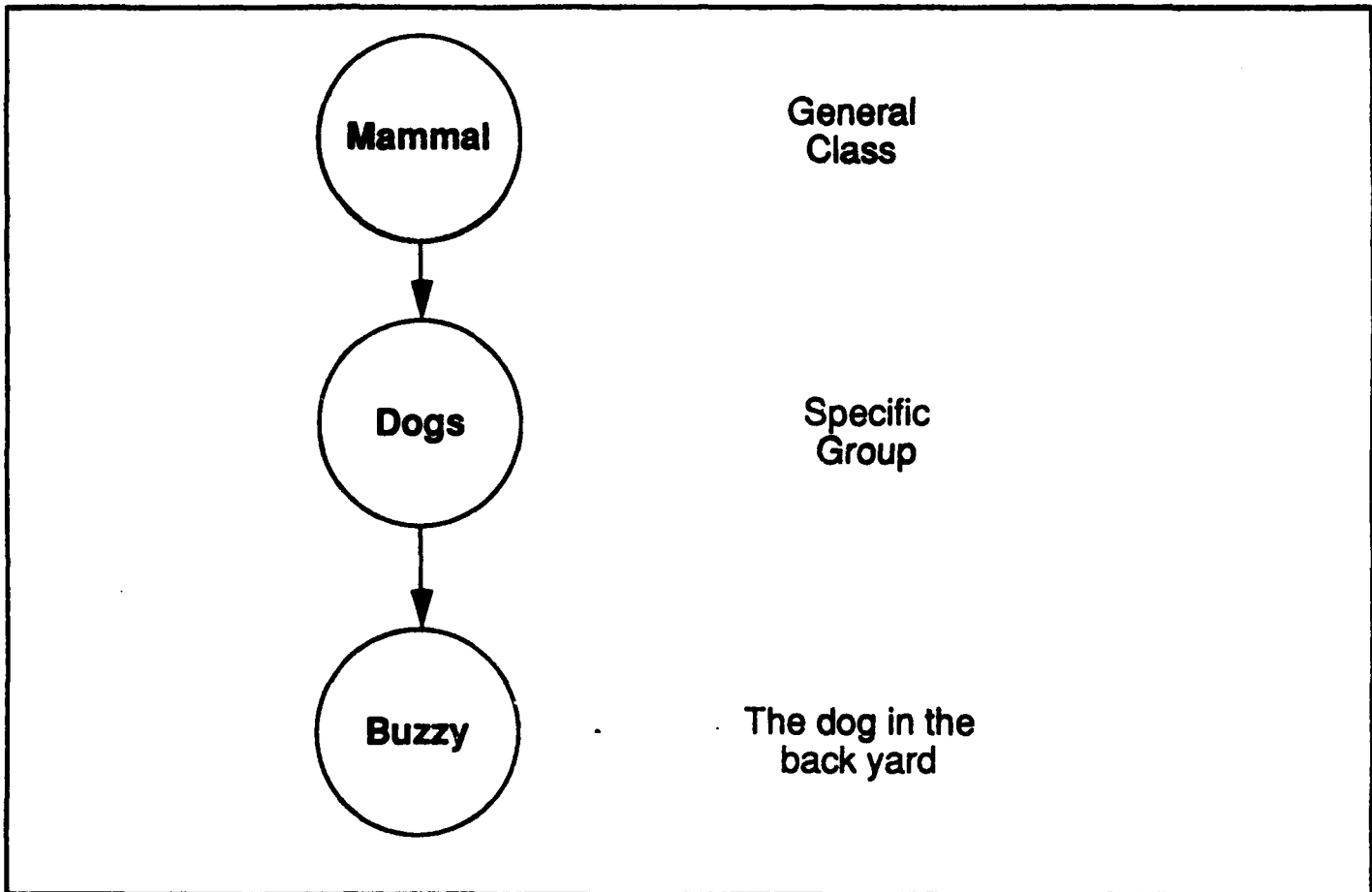


Figure 6-4 Object Hierarchy

Additionally, you should look at the documentation of the shell. Some shells have excellent characteristics but the documentation for the shell is not very good. This makes it difficult to learn hard to use the shell and when you run into problems it's difficult to find references on how to use the shell. Most shells will be changing over time so it's also important to learn what the maintenance and upgrade procedures are for this particular shell. In particular, it's important for the larger shells, which are much more expensive than the PC shells, that you know what the maintenance will be on those shells over time.

The last item, of course, is the cost of the shell. Shells range in price at the very low end from about \$100 or \$150 all the way up to \$60,000 or \$70,000. Obviously what you get at a \$50,000 or \$60,000 shell is considerably different from what you get for a

\$100 shell. However, it's also true that some relatively inexpensive shells can be highly productive, therefore, one should not buy on cost alone. It is, however, a consideration in the overall cost of the project.

6.7 Matching Requirements and Capabilities.

You will notice that in this selection chapter we have not provided a particular set of shells for you to choose from. The reason for that is quite clear. There are a large number of shells in the marketplace and it's extremely difficult to keep up with all of the attributes of each one of those shells. What we have tried to do instead is to give you some underlying principles to utilize in selecting a shell, and in particular to give you some items to look for as you make those shell selections.

We also provided some of the definitions of the vocabulary that's utilized in the shells to help you make a better decision about the kind of shell that you might need in your project. To assist you further Table 6-3 has been provided as a form that you might fill in for any shell that you might be interested in evaluating. This form will give you enough information that you can make decisions about the relative merit of the shells based upon the decision criteria given in this chapter.

Product

Vendor Name, address

Hardware Compatibility

Language Used

External Interfaces

Sensors

Databases

Other computer systems

Knowledge Representation

Control Structure

Forward Chaining

Backward Chaining

Uncertainty Factors

Truth Maintenance

Knowledge Acquisition Facility

Editing & debugging tools

Deduces rules from examples

Other

Explanation Facility

Customizable?

User Interface

Graphics

Menus

Separate run-time modules

Price

Table 6-3 Comparison Form

Chapter 7

Knowledge Acquisition

7.1 Introduction

Knowledge acquisition is the process and the essence of building the expert system. Successful expert systems are built by studying the strategies that human experts use to solve problems and then using those strategies and software to emulate their behavior. Before this system can be built, however, the knowledge acquisition must be completed. It is the most time consuming and difficult aspect of the development of an expert system project. Although recent reports show us that there are some reasons for optimism that some of the obstacles are being overcome in knowledge acquisition, the knowledge acquisition process still remains a bottleneck in the design of most expert systems.

Before we discuss knowledge acquisition more fully, it's important that we understand a couple of definitions. We make a technical distinction between knowledge engineering and knowledge acquisition in the following way.

- * Knowledge engineering is the process of understanding the structure of the problem. Understanding how knowledge is accumulated and collected in the particular application to be emulated, and understanding how the user interface should be constructed for ease of use of the system. It involves, to some extent also, some knowledge modeling process because this tells us what the knowledge base structure should look like.

- * Knowledge acquisition is the collection of the detail of the knowledge based and the formulation into the structure's design in the knowledge engineering activity.

Knowledge engineering, or the process of determining the structure of the knowledge base is often completely determined by the selection of the tool. This means that if you select a tool that does only rule based reasoning, then the structure of your knowledge will be in if/then rules. There is no other choice. This is often the case when using small tools. Using large hybrid tools, however, you often have options to utilize either objects, rules, or frames in the construction of your expert system. Which of these that you use at various points in the system will be determined by your knowledge engineering and knowledge acquisition process.

There are a number of alternative methods of knowledge acquisition available. These are documentation review, interviewing of experts, expert protocols and expert system simulation techniques. It appears unlikely that most development projects will be carried out without at least some use of documentation reviews and interviews between a knowledge engineer and the domain expert. An understanding of the range, variety and appropriateness of knowledge acquisition methods is important to each knowledge engineer. The purpose of this chapter is to focus on the four methods and to give a description of these techniques for use by knowledge engineers.

7.2 Documentation Review.

Most expert system candidates are operational with a manual process in place. In some cases, the manual may be supported by some automation. These operations generally have documentation which explains how the process is to be conducted, at least in its rudimentary forms. It makes sense to utilize this written knowledge as a basis for starting an expert system project. Obtain and review any documentation, including training class materials about the system.

7.3 Methods of Interviewing Experts.

Interviewing is the most common form of knowledge acquisition. It is the form usually called to mind when knowledge acquisition is mentioned. Every knowledge engineer must be a skilled interviewer. Some of the other techniques will also be supplemented by interviews. This makes interviewing the most important skill needed by a knowledge engineer. The following paragraphs describe several interview techniques.

7.3.1 Nondirective Interviewing. The method of nondirective interviewing was developed for use in client centered counseling and psychoanalysis, but it has found wide spread application for gaining in depth understanding of an individual's thinking process. The verbal behaviors that constitute nondirective interviewing are designed to encourage self explication by the expert and verbal statements on the part of the expert with a minimum of intrusion by the interviewer. When used effectively, nondirective responses guide and channel the interaction in a relevant and informative direction without imposing any frame of reference or biases of the knowledge engineer on the expert.

Nondirective interviewing is a very unstructured process and the knowledge engineer must learn certain tools in order to encourage the expert to reveal the process that he goes through in his decision making. These tools are in the form of responses that the knowledge engineer uses to the domain expert. The following paragraphs describe the responses that a knowledge engineer might utilize in a nondirective interview.

7.3.1.1 Continuing Statements and Behavior. The most nondirective interviewer behaviors are those which merely indicate an expectation encouragement for the domain expert to continue talking. Such statements such as, please go on, or non verbal

responses such as head nods sometimes silence can also be used to encourage continued description or to give the domain expert time to retrieve memories or to transform difficult mental operations into speech.

7.3.1.2 Reflection. Reflections are restatements or paraphrases of what the domain expert has communicated to the knowledge engineer. Use of this type of response conveys to the expert the knowledge engineer's ability to listen and understand the problem domain. Reflections are repeated information back to the domain expert, also allow an implicit accuracy check. The expert is given an opportunity to review the knowledge engineer's understanding if it is not an adequate representation of what has been said. Frequent use of reflections creates a balanced interaction in which both parties are actively engaged in exploring the problem domain and insures effective listening on the part of the knowledge engineer.

7.3.1.3 Summarizing Statements. Summarizing statements are similar to reflections in that feed back to the expert the essence of the information that has been presented. These statements generally span greater amounts of information for longer periods of time than the expert was utilizing. However, providing a recap of the points of importance in a major segment of a interview. Summarizing statements will often help to determine whether a sufficient amount of information has been gathered for a change of topic to be appropriate. If the expert indicates that the summary has omitted critical issues, then further exploration of the current topic is indicated.

7.3.1.4 Open Ended Questions. Open ended questions may promote the continued flow of talk of the expert or discourage it depending upon the way they are worded. Closed ended questions can halt the flow of description from the expert by creating a specific answer

of one or a few words. Questions that can be answered by yes or no are common examples since they allow the expert to reply with one word answers and do not encourage elaboration. Open ended questions, on the other hand, indicate a general direction for the expert to explore without shutting off the flow of description. They guide the interview while still allowing the expert to decide what content is most important. Open ended questions often take the form of tell me more about some topic, or in what other ways to you use information about some topic.

7.3.1.5 Interpretations. Interpretations are attempts to discover connections between ideas that have escaped verbalization or recognition by the expert. Often the knowledge engineer having learned about an area of the domain expert's thinking will recognize apparent aspects of the problem solving strategy which forget to be mentioned. Interpretative statements present these hypotheses to the expert for verification. These statements are not purely nondirective since they represent an attempt on the part of the knowledge engineer to uncover associations or knowledge not revealed by the expert. They are usually introduced tentatively to avoid second guessing the expert, and are considered subject to revision if they are not accepted. Nondirective interviews can be almost informal conversational interviews. They depend on the spontaneous generation of questions including ongoing observation of the expert's activities. There are no predetermined questions and the expert may not even realize the amount of data which are being gathered in the interview. In fact, if you talk with different experts different data would generally be gathered from these interviews. The major strengths of the nondirective interview is that the knowledge engineer can be highly responsive to the expert and the interview can be individualized making use of the surroundings or particular phases. The major weaknesses of the nondirective interview is that it takes a great deal of time to perform the interview. The quality of the data collected are

highly dependent upon the interviewing skill of the knowledge engineer. The nondirective interview is most often used in the early discovery process of knowledge acquisition. usually before much is known about the process to be prototyped in the application.

7.3.1.5.1. While nondirective interviewing may appear to be a relatively simple process, effective use of the method involves close attention, ongoing analysis of the content of the expert's replies and carefully worded responses. This skill is usually developed with time and experience. Without training few individuals are naturally nondirective and few are aware of how profound the effects of their verbal statements can be on the replies they receive. Performing nondirective interviewing generally requires the keeping of careful notes during the interview that will assist you as the knowledge engineer in formulating questions as the interview progresses.

7.3.1.5.2. Analysis of the data collected from the nondirective interview involves the reduction of free form verbalization from the expert into a more structured format. The techniques used for this is often called knowledge modeling. A knowledge modeling process will be discussed later in this chapter.

7.3.2 Structured Interviewing. In contrast to nondirective methods, the structured interview has a well defined predetermined agenda which guides the conduct of the interaction. Its purpose is to illicit specific data required by the expert engineer. Preparations for the interview begin with definition of the information to be obtained from the expert and formulation of a sequence of questions designed to provide the desired answers. A written interview guide is almost always used by the interviewer to assist in administration and data collection. Questions contained in this interview guide place specific non-ambiguous

expectations on the knowledge engineer and serve to control the course of the interview.

7.3.2.1. Structured interviews may be of two basic types. The interview guide approach and a standardized open ended interview. In the interview guide approach the knowledge engineer employs a list of topics to be explored along with subtopics to be probed. The issues on the guide are not necessarily taken in any order, nor is the knowledge engineer restricted to the wording of the question topics. The knowledge engineer must use his own skills to probe the long term memory of the domain expert and clarify the issues.

7.3.2.2. The strength of the interview guide is that it forms a checklist to insure all relevant topics are covered in each interview. This serves to limit to the time for the interview, as well as make the interviewing process more systematic. The weakness of the interview guide is that it destroys the spontaneity of the interview process so that a small danger exists that some relevant information might be missed. An interview guide is used in order to make sure that basically the same information is obtained from several experts when several knowledge engineers are used. This makes the interview guide very appropriate for the knowledge acquisition process.

7.3.2.3. A structured interview could use a standardized open ended interview guide with a set of carefully worded questions that are asked of each expert. This approach minimizes the knowledge engineer's effect, as it can be used with less experienced knowledge engineers. Very systematic data are obtained which makes it especially useful when there are a large number of experts to be interviewed. The weakness of this method is the knowledge engineer is constrained from probing into unanticipated topics and the thrust of an individual expert's differences are reduced.

7.3.2.4. A fundamental skill when using structured interviews is the design of questions that will effectively illicit the desired data. One concern is matching the language of the questions to the characteristics of the domain expert. Vocabulary and syntax must be chosen to maximize the accurate definition of the desired information. The goal is not necessarily simplification of language since many expert professionals will not respond appropriately to questions which are oversimplified or in "lay" language. Instead the knowledge engineer must match the wording of the questions to the communications style of the expert with sensitivity to specialize vocabularies that may be appropriate to the situation. This is the reason why knowledge engineer must have some experience in the domain of expertise for which they are building systems.

A second concern in generating questions involves the frames of reference of the expert and the knowledge engineer. Domain experts who are unfamiliar with the format and structure of expert systems software may find it difficult to respond adequately to questions which are implicitly or explicitly assuming a particular style of knowledge representation. While the frequent warning is given to knowledge engineer to avoid early assumptions about a particular scheme of knowledge representation, in practice it is often difficult to formulate questions which are totally free of such biases. Even assumptions about the generic type of the problem being solves may result in questions which the experts find inconsistent with their own thinking process.

A third issue involves the type of questions included. Each question in a structured interview should be a query about a single idea. The seemingly simple dictate is a frequently violated one. Knowledge engineers frequently fall prey to asking questions which in fact are several questions simultaneously which are difficult

for experts to interpret. In spite of its overtly directed nature, the structured interview does not necessarily have to depend upon closed end questions. In fact, it is best to avoid yes/no questions in favor of open ended questions which encourage description and discussion by the expert.

A final concern is the sequencing of questions. The interview guide should take a sensible approach to presenting questions in a sequence which is appropriate to the decision being discussed. Organization which reflects the underlying series of processing steps will improve the chances of flow and accurate report from the domain expert.

7.3.2.5 Once the interview guide is constructed administration is a process of presenting the predetermined questions or topics and following up replies from the expert with probes by the knowledge engineer which elicit further detail. In many cases these probes can take the form of open ended questions or specific closed end queries. The success of the method depends on the ability of the knowledge engineer to identify areas of missing information to generate probes which fill gaps in the data and to know when a question has been answered sufficiently.

Because of its dependence on prior familiarity with the topic material, the structured interview is typically used later in the process of knowledge engineering. The planning and analysis that go into creating such an interview may be inappropriate after the knowledge engineer has a sound working knowledge of the expert and the domain area. Careful formulation and administration of a predetermined set of questions can make the knowledge engineering process much more efficient and minimize the burden it places on the expert. This approach may also be appropriate for gathering information from additional experts who are not the central focus of the thought process.

Knowledge acquisition is a multiple step process that elicits knowledge from a domain expert. These five stages are identification, conceptualization, formalization, implementation and testing. Identification describes the problem, resources and goals. Conceptualization represents knowledge by determining key concepts and relations. Formulation involves designing the structure to organize knowledge by mapping the concepts and relations. Implementation formulates structures to embody the knowledge using rules, frames and other coding approaches. The final stage, testing, validates the knowledge structures and provides the input into the prototyping evaluation process.

The implementation of capturing this knowledge is the process of knowledge acquisition. The representation of this knowledge so that it can be understood by everyone on the team is the process of knowledge modeling. The techniques of knowledge acquisition will be discussed first, and then at the end of the chapter we will discuss knowledge modeling as a process.

7.4 Expert Protocols.

Expert protocols are techniques where the domain expert perform the task under study for the expert system and report on his thinking processes as he conducts the activity. The most common form of this activity is the thinking aloud narrative.

7.4.1 Thinking Aloud Narratives. A simple approach to discovering the strategies used by domain experts in making decisions is to ask them to label and to describe mental operations and internal thought processes that occur while they are actually engaged in solving a problem. Thinking aloud produces an ongoing narrative related to the completion of the chain of mental operations which make up decision making.

7.4.1.1. Thinking aloud narratives can produce a rich source of data for the knowledge engineer. There are actually three variations of the method. The most common is concurrent thinking aloud in which a narrative is produced during the actual performance of the problem solving task. Retrospective thinking aloud narratives which involve a report of memories of prior occasions of problem solving are also fairly common. Less frequently used is hypothetical thinking aloud which presents an imaginary situation to be solved. Both concurrent and hypothetical thinking aloud methods are actually reports on immediate problem solving processes. In the case of the hypothetical approach, the external circumstances and data are not physically present but decision processes are activated to deal with conceivable circumstances.

7.4.1.2 While most individuals are familiar with explaining how they reach conclusions, typical descriptions of this type are incomplete. Thinking aloud narratives require the expert to maintain close attention to the sequence of operations and their immediate products in his conscious mind. Many individuals utilize some form of internal speech in portions of their problem solving, and thinking aloud procedures externalize these silent language processes by producing verbal descriptions.

7.4.1.3. Thinking aloud requires recognition and verbal classification of these types of internal mental events. The knowledge engineer must pay attention to gathering a complete report of the expert's activity at each step of the decision making process: (1) the sources of external information relevant to each decision, (2) sources of internal knowledge called upon, and (3) methods of combining the knowledge. The expert will usually require some practice in thinking aloud in order to produce a sufficiently detailed thinking aloud narrative.

7.4.1.4. As the narrative is produced by the domain expert the knowledge engineer listens to the steps being described paying particular attention to obvious omissions. The knowledge engineer can then engage in cuing of the expert. Cuing is the process of prompting the expert for additional information when such omissions are recognized. Effective cuing will ensure sufficient description is obtained without interfering markedly with the expert's internal processes under investigation.

7.4.2 Expert Systems Simulation. The next knowledge acquisition approach is expert system simulation. It takes into account the fact that knowledge acquisition is often a very difficult process because the domain expert himself may not have the ability to articulate his procedures. His knowledge is compiled such that he does not understand himself how he performs this task. The first steps towards understanding how the expert performs his task can often be assisted greatly by simulating the expert system in a live experiment. The expert system simulation is conducted in a way that the domain expert plays a role of the computer expert system and a novice user is brought in to perform various case protocols. The objective of the simulation technique is to externalize and decompose the mental process of the domain expert in performing his analytical task.

7.4.2.1. The set up for the experiment includes a room with video taping facilities. The expert and the novice user are separated so that they cannot see each other such that the expert and the novice can communicate only through verbal communications. The entire process of simulation is video taped for later analysis by the knowledge engineer.

7.4.2.2. The novice has all the case documents. The expert's access to case information must be through verbal communication with the novice. This verbal communication is restricted to

multiple choice questions and answers, help request from the apprentice and directives from the expert on how to find information and perform specific calculations. As a result, the expert is forced to accurately and completely externalize the individual analysis steps and components as well as the information algorithms and rules used and assessments generated for each step associated with the particular case. In most cases several case protocols will normally be used in the simulation in order to cover as many diverse situations as possible.

7.4.2.3. Following completion of the simulation, the video tape is analyzed by the knowledge engineer. The knowledge engineers utilize the video tape to structure knowledge and to determine the thought processes that are being generated by the domain expert. This knowledge acquisition technique is often used early in the development process in order to gain insight into the structure and uses of domain expertise by the expert.



Chapter 8

Knowledge Modelling

8.1 Introduction

Knowledge modelling is the analysis process of transforming information given by the domain expert into diagrams that describe the domain knowledge with a top down perspective. The intent of knowledge modelling first, is to support the structured acquisition of knowledge and second, to provide a high level, visual graphic form to augment the detailed representation schemes such as rules and frames. It supports acquisition and representation of problem solving knowledge by:

- o Describing different types of knowledge in a systematic fashion.
- o Applying a methodical approach to knowledge acquisition.
- o Adaptability to various knowledge representation techniques such as rules and frames.
- o The use of graphic techniques to augment the scope, understanding and modularity of knowledge.

8.2 Knowledge Modelling.

Figure 8-1 represents the conventions used in developing knowledge models. The primary components of a knowledge model are events, occurrence linkages and objects. Events are descriptions of knowledge consisting of facts about actions and events that will take place, are taking place, or have taken place. Thus events have action verbs associated with them. For example, an event like "verify test indicator status" might be an event from the knowledge for equipment testing. Events are represented by boxes with a number and name to identify the event. The numbers are assigned as decimal numbers to allow for hierarchical decomposition.

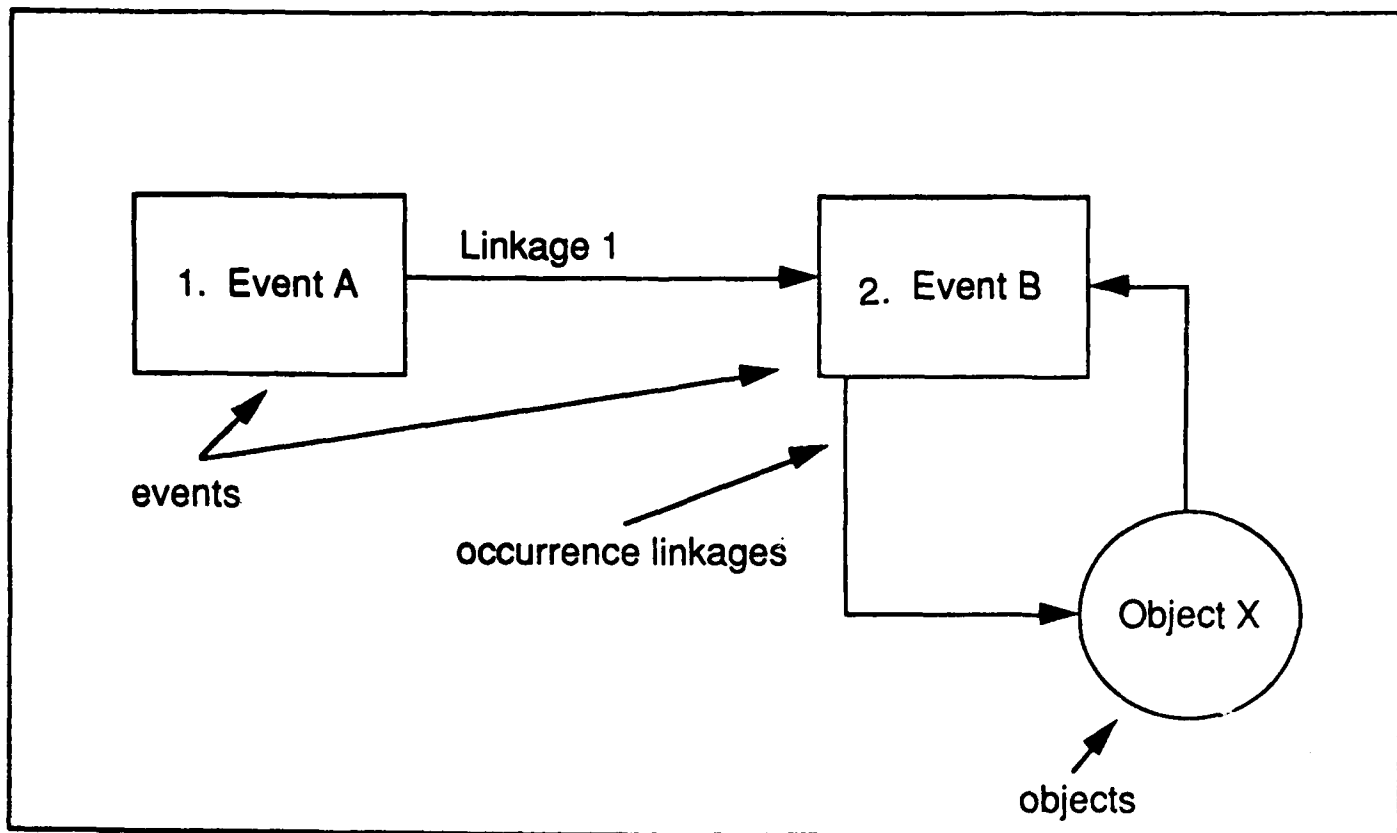


Figure 8-1 Conventions In Knowledge Modeling

Events are connected to other events by occurrence linkages, shown by directed arrows, that separate events into unique instances within periods of time. They are used to corralate the occurrence of one or more events in relation to other events. The separation of events allows them to be viewed as independent components while still realizing the overall connectivity of a system of knowledge.

Events have two characteristics that are essential for knowledge representation. First they indicate activity based upon cause and effect relationships of other events or objects. Second, events may be decomposed into subordinate events.

Occurrence linkages are directed arrows which show the direction of the knowledge flow. Occurrence may have a name associated with them by writing that name along the arrow. Occurrence linkages can go into as well as out of both events and objects.

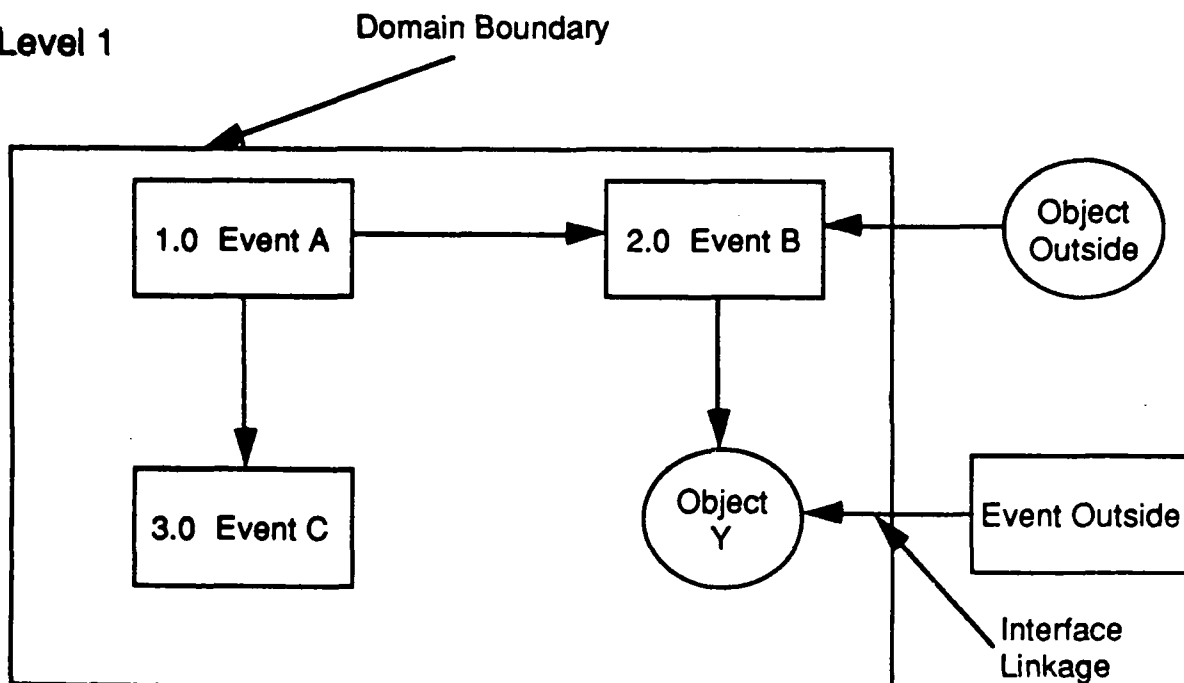
Objects are shown with their name in a circle. Objects are situation status entities and thus are usually nouns or noun phrases. Objects are used to identify persons, places or things and provide descriptions, characterizations or classifications. An object consists of an identifier to name the object, and an associated attribute such as color, type or condition. An object for the diagnostic example might be "indicator light reading is fail." The state of an object is determined by events in the diagram.

The top level view of a knowledge model is often called the domain view. This is the set of events, occurrence linkages and objects represented at the top level diagram. Events on upper levels are decomposed hierarchically into subordinate sets of events that can also be decomposed as required until a desired level of representation can be reached. An event at the lowest level of the diagram is called a primitive event. A primitive event is determined when it is no longer practical or productive to decompose an event further.

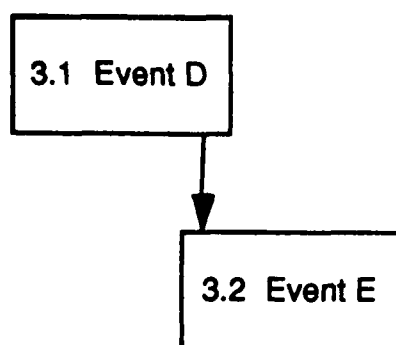
The levels of a knowledge model are identified by a number assigned from the top down. Figure 8-2 shows the levels typically used in knowledge modelling. Level 1, shown in 8-2A, is also called the domain view for a knowledge model. The box around the Level 1 diagram shows the extent of the domain. The need for the boundary is to show where outside events influence the domain. For example, ambient temperature and humidity conditions may influence diagnostics, but are not part of the system. The linkage connecting into the domain from outside is an interface linkage.

The Level 1 diagram is decomposed into a level 2 diagram by breaking the level 1 events into their components. Figure 8-2B shows a Level 2 diagram by decomposing one of the events from Level 1. Level 2 diagrams are then decomposed into Level 3

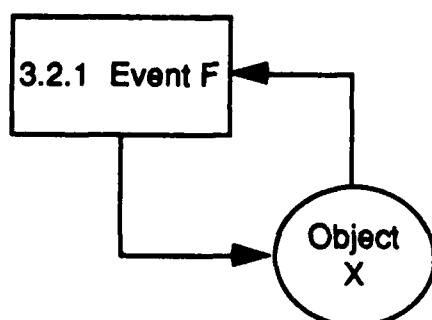
A. Level 1



B. Level 2: Decomposition of Event C



C. Level 3: Decomposition of Event E



By definition,
primitive events are
at the lowest level
of diagram.

Figure 8-2 Knowledge Model Levels

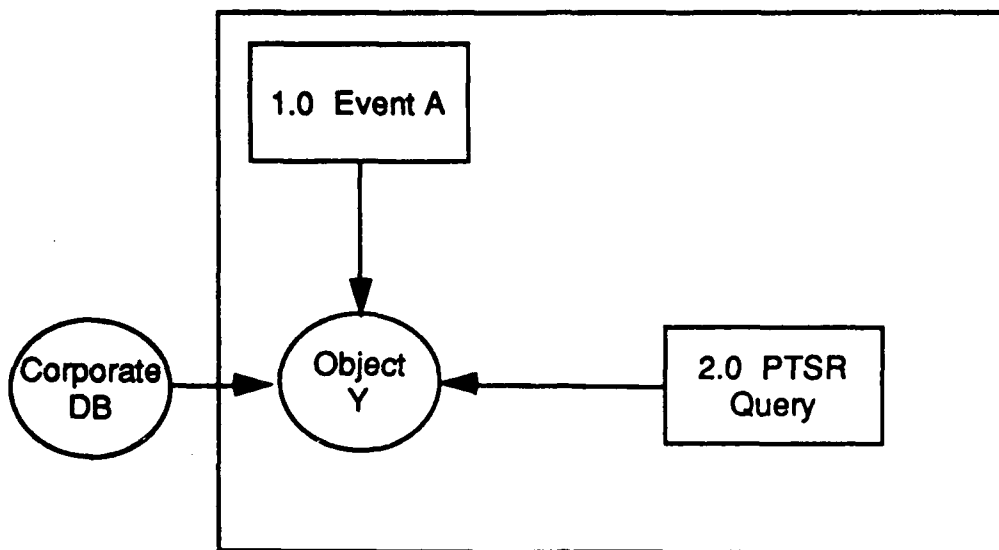
diagrams. For example, Figure 8-2C shows a level 3 diagram for one event in the Level 2 diagram. You should avoid going below Level 3 in your knowledge models. Lower levels typically add complexity beyond the understanding of most individuals. If you find going to lower levels is necessary, try to redefine the domain into several domains that interconnect at the top level. This will usually simplify the domain to only three levels. Figure 8-3 is an example of a knowledge model from the MOBPlex development.

The graphic model of a knowledge model is not complete without some description of the manner in which the knowledge is manipulated. These descriptions are written at the lowest level of the diagram for the primitive events, objects and linkages. The descriptions are written in a form of structured English using action verbs, qualifiers, quantifiers and sequencing words. Formally, this is called a Knowledge Representation Language (KRL). Several KRL's exist and have been defined.

If you know a KRL, then use it. If you do not know one, it is relatively easy to create a simple one of your own for your use on the project. For each of the bottom level events, objects and linkages write a definition of terms and a description of the activities. Do this utilizing a structured English language with short unambiguous statements about the events. These statements should be very specific and limited to the task being described. You should use short structured sentences, leaving out pronouns and modifiers of the English language and sticking to action verbs and the names of the entities that are being described. An entry for a diagnostic problem might look like this:

```
Rule 3.2.1.    IF indicator light status is fail
                THEN check input voltage on pin 42
                ELSE verify start switch is on.
```

Level 1.



Level 2.

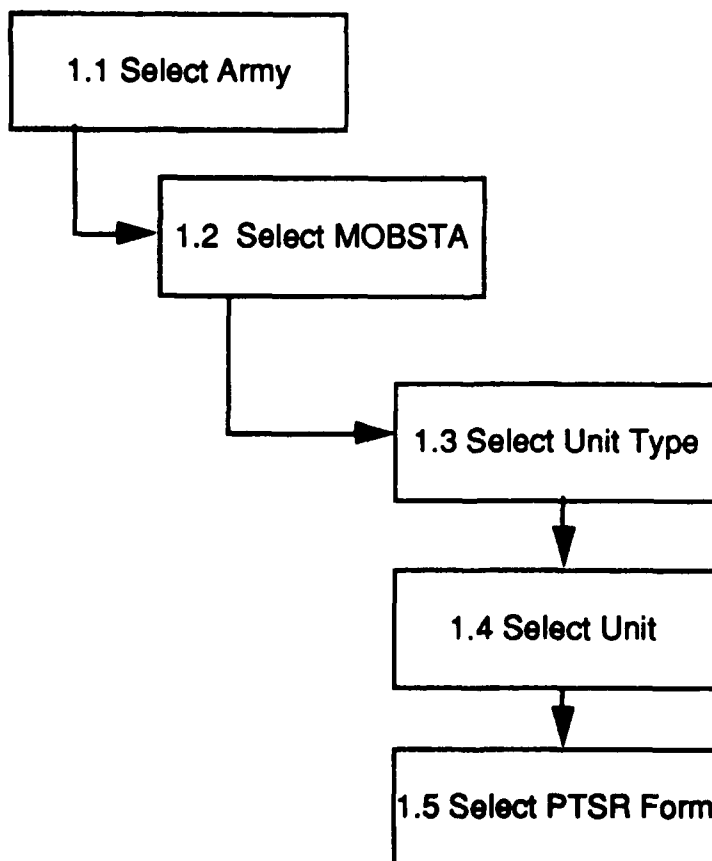


Figure 8-3 Example from MOBPlex Knowledge Model

The language you create will be essentially comparative verbs, events from the knowledge model, and qualifiers and quantifiers for the objects. Those three classes of words should be used in your KRL. The sequencing of entities is provided by the sequencing words. The sequencing words are: if, and, then, while, else, otherwise, and until. These sequencing words provide the set of operations and conditions under which particular events can take place.

You should also recognize that in some processes there will be a need for algorithmic or mathematical equations. Equations are very concise and a precise way of stating certain general principles. They are absolutely necessary, of course, when you are working with numerical data and one must perform calculations. Just because you are building an expert system does not mean that you have to shun the use of algorithmic computations. In many cases, these algorithmic computations are a part of the decision process. When the process calls for it, you should utilize algorithms and equations in precisely the same way that you would in any traditional program.

Some hints for building the knowledge model are to first of all assume that the knowledge model will have to be redone. You will typically not create a correct model on the first attempt. The following points will assist you while you're creating the model.

- * Name knowledge flows first.
- * Minimize the linkages (regroup when necessary).
- * Make notations of questions and assumptions that you make as the model is being built.
- * Do it over again.

One last point of the knowledge model that is extremely important; the knowledge model is not a flow chart, and while it does in some

ways look a bit like a flow chart, this is not a flow chart for a computer program. Rather, it is a description of how knowledge flows inside the organization. Often knowledge follows the flow of data around the organization. Therefore looking at data flows is helpful in creating the knowledge flows. But it must be imperative that you do not get this confused with a flow chart. The knowledge model does not show program controls. That's a very important point. What we're concentrating on here is the information flow, the knowledge flow and the processes. You are describing the decision making environment. The last point is to remember that only the primitive events, or the lowest level events, are the ones that are described in the KRL.

8.3 The Modelling Cycle.

The same knowledge modelling techniques can be used to produce different views of the same decision environment. Figure 8-4 illustrates the kind of modelling cycle that we might go through. We begin with a current physical model in the lower left hand corner of the figure. Through a process of discovery where we interview experts or observe their behavior. We move from the current physical model and create a logical model to describe the process.

Through a process of design, we move from a current logical model to a proposed logical model. From the proposed logical model we go through a process of implementation: the process of developing, designing and building the system. Building prototypes, cycling back through prototypes, creating the software, will move us from the proposed logical model to the proposed physical model. When we move from the proposed physical model to the current physical model back to the place where we began again, we transition this system into the user environment. That occurs utilizing user testing, integration with the existing hardware and software, validation, verification, user training, and maintenance.

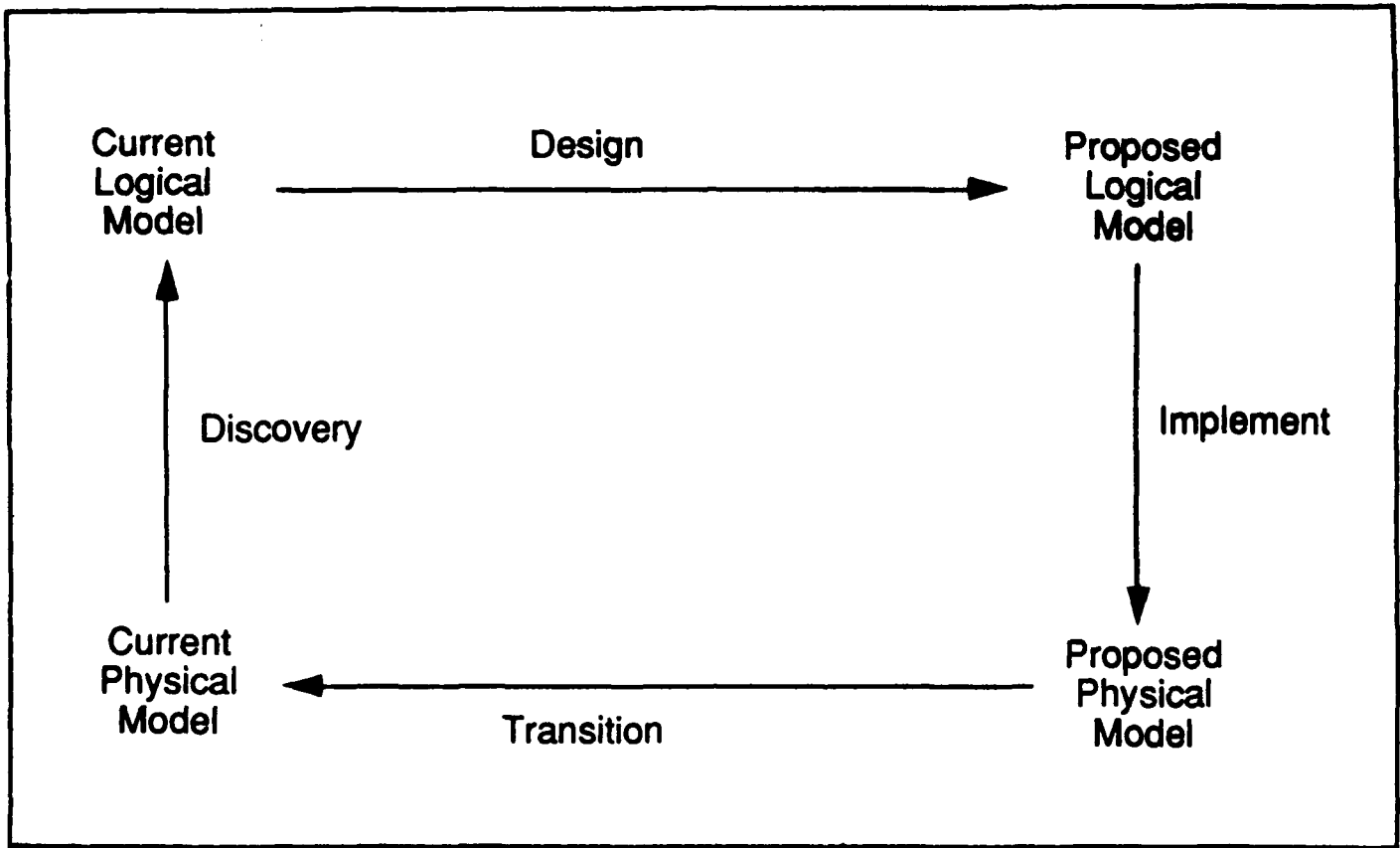


Figure 8-4 Modelling Cycle

Chapter 9

Design and Prototyping

9.1 Introduction.

At the conclusion of knowledge modelling, the development team will have defined the functional breakdown of the expert system, it will have defined its problem solving behavior and all of its interfaces that are required to support the system. These interfaces will include user inputs as well data base interfaces or interfaces with other systems that are necessary to support the expert system itself.

9.2 Expert System Design.

In the expert system design phase, we will be primarily looking at the user interface design based upon the shell that we have chosen to use. The phases in this user interface design is to look at the user perceived objects, the display layout, the interaction sequences and output behavior. At this point we have broken the problem solving function down into tasks small enough so that we will be looking at individual user acts and the expert system responses. At this stage of the process we are specifying interactions that will be at the level of detail of the actual instruction for the user to operate the system.

9.2.1 User Perceived Objects. The system should use objects in its operation that the system user recognizes. These user objects should be in the context and vocabulary of the using environment. Both the designer and the user must share the perceptions of each of the mentioned objects in the user interface and there may also be unmentioned objects in the context that they must also understand.

User perceived objects are of several kinds. First, there are data objects that represent real world entities such as vehicle type or amount of classified material. There are data objects that represent attributes of the entity such as UIC, type, or amount. There are display objects of many kinds. Some of these display entities, for example might be a menu or a graphic picture of a movement or an item inside the unit. It might be a table, it might be a graph. As these examples show a display object can represent an entity in different ways. It might have an appearance that reminds the user of the actual entity, it might display the value or the key attribute of the entity or display the values of several of the entity's attributes, it might also compare entity attributes to other entity attribute values.

In addition to display objects that represent entities and attributes of the problem model there are objects conceived by the designer to manage user interaction. For example, the user will be aware of the main window of the screen. The system will also have pop up windows that exist when requested by input from the user. These windows and menus that pop onto the screen are designer defined objects. They are manipulated by the user during the operation of the system. All of these objects and the way in which they are used must be designed by the developer of the system.

The knowledge model will help in designing names and descriptions of all the objects involved in the set of tasks for the domain. This set of tasks will comprise the problem solving functions as defined in the knowledge model. The process of relating the problem solving functions to the actual screens, objects, menus, graphics that the user will see is the process of designing the user interface.

As you design the user interface, you should keep the following underlying principles in mind.

- * Interface should be transparent - the user should be able to focus attention primarily on the problem objects rather than on the display that represent them.
- * The interface should be a user documented in the user manual in terms of the user perceived objects the designer creates, because the user is in fact going to work with these objects in the system not the real thing.
- * The interface must be system documented in terms of the created objects without any reference to the real world. When the functional design is translated into the program, nothing can be left to the computer's imagination, experience or judgement.

9.2.2 Design Display Layouts. The next step after creating user perceived objects is to create display layouts. This design process includes the definition of the architecture of the screens and the layout of each screen. These layouts are the places to put the objects. The user can recognize them and interact with them.

At the highest level are the objects called screens. They are laid out in a structure called the screen architecture. A screen is everything the user sees on one monitor at one time, plus those things that might be popped up in windows or scrolled to. The screen architecture should follow naturally from the functional architecture. Each screen should support one high level function or separable part of one high level function, or a collection of related high level functions. Figure 9-1 illustrates a screen architecture of MOBPLEX by showing the screen sequences for a typical session of querying the data base.

* Screens Querys Reports Updating Utilities

***** Postmobilization Training and Support Requirement *****

1st Army
2nd Army
4th Army
5th Army
6th Army

* Current Location *

PTSR Cabinet

Mobilization
Planning
Expert System

* Doors *

* Screens Querys Reports Updating Utilities

***** Second Army *****

Camp Blanding	Camp Shelby
Ft. Benning	Ft. Bliss
Ft. Bragg	Ft. Buchanan
Ft. Campbell	Ft. Chaffee
Ft. Dix	Ft. Eustiss
Ft. Gordon	Ft. Hill
Ft. Jackson	Ft. Knox
Ft. Lee	Ft. Leonard
Ft. McClellan	Ft. Polk
Ft. Pickett	Ft. Redstone
Ft. Riley	Ft. Rucker
Ft. Stewart	Ft. Story

* Current Location *

2nd-Army Drawer

Mobilization
Planning
Expert System

* Doors *

PTSR

Figure 9-1. MOBPlex Screen Architecture (page 1 of 3)

* Screens Querys Reports Updating Utilities

***** Ft. Stewart Mobilization Station *****

Artillery Units
 Aviation Units
 Cavalry Units
 Medical & Hospital Units
 Engineering Units
 Infantry Units
 Maintenance Units
 Military Police/Intelligence Units
 Ordnance Units
 Quartermaster Units
 Signal Units
 Supply Units
 Transportation Units
 Other Units

* Current Location *

Ft. Stewart File

Mobilization
 Planning
 Expert System

* Doors *

2nd Army
 PTSR

* Screens Querys R

Fort Stewart Infantry Units.

1/122D INF (TLAT)
 HHC 485H INF BDE(M)
 HQ 2/121ST INF (M)
 1ST BN 121ST INF (M)
 4TH BN 118TH INF (MECH)
 1ST BN 118TH INF (MECH)

nce Units

Quartermaster Units
 Signal Units
 Supply Units
 Transportation Units
 Other Units

* Current Location *

Ft. Stewart File

Mobilization
 Planning
 Expert System

* Doors *

2nd Army
 PTSR

Figure 9-1. MOBPlex Screen Architecture (page 2 of 3)

• Screens

Select a Form for Unit 1ST BN 121ST INF (M)

General Information	Training
Intelligence and Security	Logistics
Personnel and Administration	Medical
Automatic Data Processing	Legal
Communications Electronics	Dental
Remarks	

• Scr

Please select the fields for the General Information report.

☐ Unit Identification Code
☒ Unit Name
☐ Section
☐ As of Date
☒ MTOE/TDA Number
☐ E-Date
☐ DODAC
☒ Home Station St.
☒ Home Station City
☒ Home Station Zip Code
☐ Mailing Address St.
☐ Mailing Address City
☐ Mailing Address Zip Code
☐ Mob Station
☒ Supporting Installation
☐ Mob Site
☒ Capstone

OK

Figure 9-1. MOBPlex Screen Architecture (page 3 of 3)

Design the screen architectures instead of simply letting it grow. The screen architecture is a good one if it is easy for the user to remember where he or she is at all times, and easy for the user to remember where to go at all times. Avoid the proliferation of adjunct screens. It confuses the user who must work with the screen that is under a screen that is under a screen.

9.2.2.1 Screen Architecture. In the screen architecture design, allow the user to go freely from any screen to any other screen that is immediately superior to it, immediately inferior to it or parallel to it. Do not force the user to do things in an arbitrary way.

In collecting data for use in the expert system, no one kind of data should be required to be collected before any other kind, even if the second data are dependent on the first. For example, in a scheduling expert system you cannot declare precedents in activities until the activities themselves have been declared. However, the expert system can allow the precedence declarations and then either provide an error message that activity declarations must eventually be done, or supply default declarations if the user selects that option. The basic principle is not to force a sequence of operations directly on the user because it is not necessary to do so. Traditional programs generally force a specific sequence on the operation of the system. Expert systems should have a sequence determined by the case data entering the system. Sequence restrictions benefit only the programmer and not the user.

Do not try to use screen architecture restrictions to make errors and inconsistencies impossible. Instead, design the system to react gracefully to errors and inconsistencies. A good expert system can recover from any inappropriate data, so it is not necessary to channel the user narrowly. In fact, one of the uses

of expert systems is sometimes the entry of inconsistent data in the process of performing what-if modeling.

9.2.2.2 Screen Display Layout. After screen architecture is designed, the next task in creating display layouts is to design the layout of each screen. This is where the prototyping tools of expert systems are extremely valuable, because they allow us to create screens interactively that are operable as an actual system rather than as dummy screens. As you create these screen layouts, you can create the location of windows on the screen and objects within the windows.

If the screen architecture calls for too much to be crowded on a single screen, you will often find yourself unable to work on that screen. Do not attempt to put too much information on a single screen. You will begin to understand you have too much on a screen when you are forced to account for very small pieces of space on the screen, or to worry about whether or not an object can be placed on a screen or it has to be revised in size in order to get it on the screen. During this prototyping phase of building the screens, screen crowding will become rather naturally obvious as objects come on and off the screen.

The prime tradeoff will always be in the screen architecture and the screen layout, the disadvantages of putting too much on one screen against the disadvantage of using several screens. The ideal for any high level function or closely related group of functions is one huge screen displaying everything at a glance. Unfortunately current hardware/software limitations do not allow this. Generally you start with the notion of supporting a function with one screen and then you realize that not everything will fit.

The next step is to investigate space conserving strategies.

- * Crowding, smaller type, less white space, more abbreviation, tighter layout
- * Scrolling
- * Pop-up windows and menus
- * Zoom and scale change
- * Additional screens

Crowding can save only small amounts of space. It makes for a less effective interface and also has the disadvantage of locking the designer into an inflexible design with petty limitations on the characteristics on display objects.

Scrolling can save more space. If scrolling is needed on a window, it should be provided as a general utility for all windows on the expert system. Vertical only scrolling is reasonable when objects occupy horizontal zones all the way across the window. Horizontal scrolling is reasonable for time scale displays where time runs along the x-axis and the user is normally interested in the entire time span, such as in scheduling systems. Scrolling should also be combined with a facility for altering the display order of objects, unless there is a natural order that is always appropriate, such as an alphabetical order. If the user can gather together the most important objects, the disadvantage of having other objects off the screen is reduced.

Pop-up windows gives the user direct control of temporary display of material of temporary interest. If pop-up windows are needed anywhere in the expert system, it should be provided as a general utility for all windows in the system.

Zoom is useful for space manipulation where the display is two dimensional and there are meaningful tradeoffs between seeing detail and seeing surroundings. One dimensional scale change is also possible, but rarely used.

Additional screens have large amounts of space. The main disadvantage to the user is that related information that is not carried over to the new screen is far away in the sense that there's significant user effort to go back and look up a forgotten data point from the previous screen. There's also danger of disorientation. For the designer, additional screens must be named and documented separately and they contribute to complexity.

9.2.2.3 Types of Screens and Windows. Screen layout should follow a generic pattern. Typically, a screen has three permanent windows.

- * Main window
- * A control or menu window
- * A message window

In some designs the control window or main window can be temporary and might be a pop-up window only. This is rare except in a very small system. The main window handles the functional interactions with the user works with specific part of the problem that the screen handles. The control window handles interactions which the user is controlling how the expert system operates. The message window displays information to keep the user oriented and informed. These are explained in more detail below.

The main window displays what the user is working with. For example, if the user is adjusting a schedule, the schedule is in the main window. If the user is solving a location problem a location map is shown in the main window. If he is editing a report the report is in the main window. For example, Figure 9-2 shows a main window in the MOBPLEX expert system.

The control window mediates user interactions in which the user is working with the system rather than with the problem. Screen exit is a good example of a control command that belongs in a control window. Figure 9-2 shows the main control window on the upper line

of the main window. If the interactions in the main window are highly pointer oriented, there should be control commands in the control window to govern their behavior. For example, in the form for extracting information about security storage in MOBPLEX, the user selects options from the form window and then exits that window as shown in Figure 9-3. With pop-up window techniques, control windows need not be large. For example, if the user can go to many different screens from the current screen, a command menu item could be provided in a pop-up window with a menu. If the user selects a menu of other screens can also pop-up. An example of this was shown in 9-1 from the MOBPLEX system which shows a sequence of selecting the army, the mobilization station and the subset of units at the mobilization station.

The message window is a place set aside with a permanent display of such things as the name of the problem, the name of the current part of the problem, recent history of transactions and such things as error messages and prompts. In most expert systems, including MOBPLEX the system is always either ready to accept a certain kind of user input or it is not ready to accept any input. Thus, some prompts, such as touch or type or wait should be placed on the display at a fixed place in the message windows. MOBPLEX message windows are at the bottom of the screen such as shown in Figure 9-2, lower left corner, PTSR Cabinet.

9.2.3 Interaction Sequencing. The next element of user interface design is the specification of interaction sequences. Each function that the user can perform or can cause the expert system to perform will have already been broken down into a series of user input acts and expert system responses to those acts. For each screen it is now necessary for you to list all the functions that the screen manages. In the prototyping phase (see paragraph 9.3,) you move from designing the function management into building the system to perform the functions.

* Screens Querys Reports Updating Utilities		
***** Postmobilization Training and Support Requirement *****		
1st Army 2nd Army 4th Army 5th Army 6th Army		
* Current Location * PTSR Cabinet	Mobilisation Planning Expert System	* Doors *

Figure 9-2. MOBPLEX Main Window

* S

Uni
Add
Uni

Please select the fields for the Intelligence & Security report.

- ☒ Unit Identification Code
- ☐ Section
- ☐ Classified Material Accompanying Unit
- ☐ Linear Feet
- ☐ Additional Storage Requirements
- ☒ Additional Storage Feet
- ☐ Authorized COMSEC/CRYPTO Equipment
- ☐ On-Hand COMSEC/CRYPTO Equipment
- ☒ Unit COMSEC Account Number
- ☐ Confidential Security Clearance Processing Required
- ☐ Confidential Security Clearance Processed
- ☐ Secret Security Clearance Processing Required
- ☐ Secret Security Clearance Processed
- ☐ Top Secret Security Clearance Processing Required
- ☐ Top Secret Security Clearance Processed
- ☐ Map
- ☐ Map Not
- ☐ Required

OK

Figure 9-3. MOBPlex Pop-up Control Window

9.2.4 Output Behavior. An expert system does not physically operate objects and elements. Its final results are a report that the user has worked with (perhaps proved) and in some cases the output could be updating transactions to an external data base. Either the report or the transaction is the only output interface that most expert systems provide. Normally the report produced will be a column of numbers, an English language report, or in some cases graphic output that can be easily understood by other users.

As the information environment becomes more sophisticated, reports that can only be read and understood by human beings will become less useful. A decision is useless unless it is implemented. Implementation will begin to involve entry of decisions into connected information systems and it makes little sense for the results of one system to be manually copied by human beings into another system. Therefore, in the design phase you must make allowances for integration of the expert system both from the standpoint of taking input as well as putting output back in to the main stream data processing environment.

The major question to be asked in taking output from an expert system is "Who or what is the report for?" If it is for input into another system there need be no elaborate communication linkage. The report should simply be designed so that it is readable into the destination system or into a translator program that can convert it.

A semi-automated destination alternative is that the result will be incorporated into another report. For example, MOBPlex provides output into the Lotus spreadsheet as a semi-automated destination. From the spreadsheet the user of the system can design and build various ad hoc reports that are useful to the user of the system or the commander who's made an ad hoc request. This also makes the report completely editable and transferrable to other systems.

Screen dumps should never be used as text output. Only rarely would the format of a screen presentation be suitable for hard copy, since screen displays utilize severe space saving techniques. Paper output will normally give much more detail than a screen.

For example, the MOBPLEX system is designed to give day by day reports on logistical materials such as fuel and ammunition as well as personnel as shown in Figure 9-4. Making a day by day report of this type would be extremely difficult on the screen, but is relatively straight forward as a paper output report.

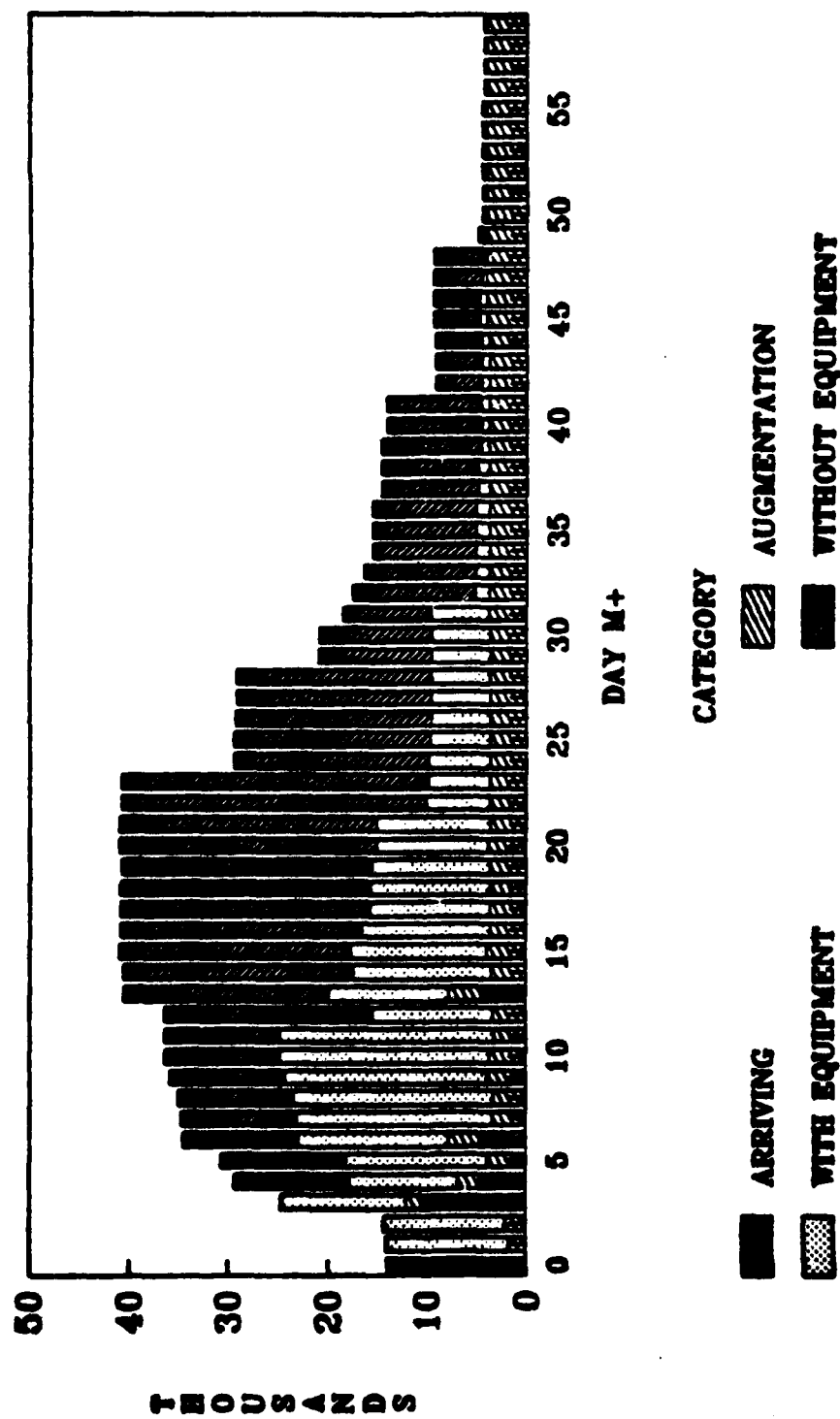
MOBPLEX includes both standardized report outputs as well as ad hoc reports. These reports were designed on top of the Lotus 1-2-3 interface. Lotus was used because it was decided there was no need to build a powerful ad hoc report generator when one was already available.

The expert system should also allow for deflection of output from the system to go into a file. This file will save the results of the system and the case information as well, so that the case can be recalled at a future time. The recall of a case is to look at the input as well as the output that was produced by the expert system.

The input/output file will be utilized for three purposes. First, it can be utilized in testing, to see if the system reproduces the same output given the same input. Secondly, the input output case material can be analyzed by other experts, human experts, who will validate the use of the knowledge base. The third reason is that in some cases we will want to revise case input of a previous case. It makes no sense to completely re-enter the case. Rather we would call up the previous case, make our modifications or perform our "what if" modelling, and then continue. We might even want to save that case as another case in the system.

DAY BY DAY REPORT

MOB-STA: FT. STEWART



NUMBER OF PERSONNEL ONBOARD

Figure 9-4. MOBPlex Personnel Loading Graph

9.3 Prototyping Phase.

The prototyping environment provides numerous useful utilities that will be used by the programmers in order to translate the design features into a working prototype. Design and prototyping is a highly interactive process, in fact, the process of knowledge acquisition, design and prototype occur in a sequence. The sequence, however, is repeated numerous times throughout the design and build phase of the project.

Prototyping is merely a way of saying that we're going to build a portion of the system at a time, and have that portion of the system be completely operational. It will not have all the features of a complete expert system, but those features which are there will work completely.

Because prototyping is continuous and iterative the final design of the system is never completely frozen, but rather the design is revised and worked out through the prototyping process. The implementation of the code in the software therefore, must be supportive by powerful programming environments. These powerful programming environments or expert system shells, provide the assistance to a programmer to allow him to literally take information from an expert and encode it almost as the expert is speaking.

The design of the expert system will grow and evolve during the prototyping phase, no matter how detailed and how careful the design has been done in the previous phases. New problems, new modules, new requirements will arise during the prototyping phase. Programmers will continue to isolate objects and procedures throughout the prototyping procedure. The prototyping phase will require good coordination throughout the team, between the knowledge engineer, the programmer and the domain expert. It will

require a lot of references back to the knowledge acquisition phase and the knowledge model as well as the system design.

Generally speaking the prototyping phase will be conducted in a series of prototypes. Usually prototyping phases are approximately 90 days in length, which is to say the sequence of knowledge acquisition, design and prototyping phase occur over an approximate 90 day period. At the conclusion of that 90 day period, the running prototype will then be presented back to the domain expert in order to obtain his comments. The comments from the expert will then be utilized as a basis for entering into another knowledge session. From this knowledge acquisition, more design can take place, and another prototype. Another 90 day sequence of events.

At the end of the second prototype again the prototype is presented to the domain expert and the sequence is begun again. The procedure for evaluating the expert system by domain experts will be discussed in the evaluation methodology, which will be contained in Volume 2 of this series.

Chapter 10

Implementation Strategy

10.1 Introduction.

The implementation of the expert system into the organizational environment can be very difficult. Most likely, the expert system will cause a change in the behavior of the users of the system. The most obvious of these changes is that now the user will have to interact with the computer in order to get his job done. Some individuals accept this interaction with the computer differently from others. For this reason, it is necessary to have an implementation plan to effect a smooth implementation. If you have built the best expert system ever conceived by man, it will provide no benefits unless it is placed into operation by the target user group. A good implementation plan is the difference between complete, enthusiastic acceptance of the system versus resistant, grudging use or even outright sabotage of the system.

10.2 Managing the Implementation.

Any technological change will be resisted by the organizational culture. This is a normal human reaction to environmental change and is to be expected. Most individuals would prefer to keep the status quo than change it. The expert system (or any other computer system) will change behaviors. The most important factor in gaining acceptance of the expert system is to design a system that fits the organizational needs. This is the reason the development team selection placed a great deal of emphasis on insuring users were part of the development. The using community will be much more likely to accept a system they had a hand in designing. This is the basic premise of the stakeholder assumption discussed in Chapter 5.

The resistance to technological change is rooted in the organizational culture. A culture is defined as:

Organizational Culture - a pattern of basic assumptions invented, discovered, or developed by a given group as it learns to cope with problems of external adaptation and internal integration that has worked well enough for the organization to remain viable.

The organizational culture exists in every organization. It is the way the aggregated group responds to its work place. This organizational culture has three levels: surface level, subsurface level, and hidden level.

The surface level of the culture is defined in the artifacts of the organization. These include any observable behaviors including slogans and logos of the organization. Slogans and logos have been rallying symbols for organizations, especially military ones for centuries. For example, contrast the feelings you have when contemplating 7th Army's patch of "Seven steps to Hell" with the logo of a Navy fighter squadron known as the "Tomcatters" showing a wild looking tom cat carrying a bomb with its fuse burning. Very different images are usually created in an individuals mind when he sees these logos. These symbols are part of the surface culture of the organization.

The subsurface level of the culture is defined in the value structure of the organization. The values can be identified only by probing beneath the surface of the organization. The language and vocabulary of the organization will often give clues to the subsurface level. For example, when the organization is performing its tasks manually, it will use the terms appropriate to the manual process. As the expert system is accepted, the group will adopt the new vocabulary to describe its actions.

The lowest level of the organizational culture is the hidden level. It is hidden because even the members of the organization usually

do not realize these parts of the culture exist. The group members would require some outside assistance in order to identify the attributes of the hidden culture. Indeed, the knowledge acquisition process described in Chapter 7 has as its focus the deep probing to determine what the artifacts of the hidden behaviors of the domain expert are. These might also give clues to the hidden culture.

Introduction of the expert system will cause change in much of the culture. The most immediate change will be the use of the system by the members of the group. This is a surface change and one that can be implemented by directive. This does not mean, however, that the group will fully accept the system. Full acceptance of the system will be effected only through the subsurface changes that cause changes in the value system of the group.

Subsurface changes will be required for full acceptance of the system. These changes can only be effected by approaching the implementation as a problem in conflict resolution.

10.3 Implementation Plan.

An Implementation Plan must be executed in order to achieve orderly phase in of the expert system. The plan is written to approach the implementation as a conflict resolution problem. Actions in the implementation plan are directed at inducing behavioral change at two levels of organizational culture change: surface changes and subsurface changes. Surface changes can be achieved by directive: requiring use of the system under defined circumstances. These changes can be implemented immediately. Subsurface changes are much more difficult to achieve because they require a change in the group value system. These changes require more time to implement because adjustments must be made and internalized by the entire group. Implementation actions are divided below into those directed at surface changes and subsurface changes.

At all times, it must be remembered that implementation is a process that must be managed. It is not sufficient to publish a guide for use of the system and a time-phased implementation. The way in which the implementation is managed will determine whether the implementation is one of enthusiastic adoption, persistent grudging use, or sabotage. The probability of enthusiastic adoption can be raised by an open communication environment and lots of feedback, both up as well as down the line.

10.3.1 Surface Implementation Actions. These actions will change the immediate artifacts of using the expert system in the department.

- * Phase-in the system by assigning a unit to adopt the system over several months. Start with the unit that is most positive toward the system and work toward the most negative. The length of time required for the phase-in will have to be determined by the strength of user acceptance of the system and by organizational need.
- * Allow the manager of the using unit to control the implementation. Giving the manager this autonomy will also help in effecting subsurface value changes that are required for the ultimate full acceptance of the new methods implied by the expert system. Each manager should discuss his plan with his supervisor to obtain approval of his/her plan.
- * Each user must be given full training before using the system. User support must also continue beyond formal training. The training must include some of the rationale for the system and insight into how the system works.
- * Publish a memorandum defining explicit guidelines for use of the system.
- * Publish a checklist of data required to use the system.

10.3.2 Subsurface Implementation Actions. These actions are intended to change the shared group values about the use of the expert system in the organization.

- * Establish a management forum to discuss cases or situations entered into the system. The user responsible for the case/situation could present it to the forum. Discussing the cases will show the value added by the system as well as initiate subsurface structure changes.
- * Indicate how results of the system will be used in the decision making process. If possible, define some guidelines for when the user can go against the system based on external contingencies.
- * Publicize results of the system. Communication is key to counteracting rumors about the system results and uses of results.
- * Listen to users. Their interaction with the system can provide good points for future versions of the system. When users see their ideas put into the system, they identify with it more strongly thus moving their value system closer to that implied by the expert system.

Chapter 11

Documentation and Maintenance

11.1 Introduction.

The documentation of an expert system is all the materials both text and graphs produced by the expert system development team that serve to explain to users how the expert system behaves and to system developers the detail structure of the expert system. Because there are two audiences users and developers there are two broad classes of documentation; user documentation and system documentation.

This chapter treats documentation in detail, with emphasis on documentation that is delivered with the finished system. Early stages of documentation, including the brief description that documents expert system ideas for screening and the functional description that documents conceptual design are covered in the first part of this chapter. The chapter will also provide guidance for requirements definitions, system specifications and design when they are required as well as a User Manual for the system. Documentation as defined above includes several items that are not always recognized as documentation: the source code, when carefully annotated with comments is the primary document of system documentation; annotated lists of error messages, help messages, menu items, commands, variables, data dictionary, and so forth; diagrams of screen layouts; documentation of utilities not written by the developer (compiler, operating system, and so forth); and internal development documents such as standards for programming practices and development methodology.

This chapter also discusses documentation for a third potential audience - potential sponsors or superiors and a brief treatment of proposals contained in paragraph 11.5.

11.2 Elements and Phases of Expert System Documentation.

Figure 11-1 lists the documents of an expert system, shows their order of preparation and relationships to each other. The first document that describes an expert system is a brief description no more than 2 pages long that documents the initial concept for the expert system and is used for initial discussions and comparisons the concept with other competing concepts. This process allows the decision making team to determine if the expert system concept warrants further development. The following paragraphs show how to prepare a brief description.

11.3 Brief Description.

It is generally impossible to tell exactly where the concept for an expert system begins. Generally the most important aspect of beginning a concept is a user need that is exhibited in the user organization. This need is then articulated as a problem to a potential builder of the expert system. These early discussions are the initial concept development stage and are documented in the form of brief descriptions, in an attempt to launch an expert system development project that would be needs driven as opposed to technology driven.

For example, the MOBPlex expert system ideas came from active army units and mobilization planners that have problems in dealing with data base access for mobilization data. The intent was to create an expert system idea from the viewpoint of what decision aiding opportunities existed, rather than from the standpoint of what expert system resources were available. The process of articulating needs and giving an early design concept is a process of high level synthesis. The output from high level synthesis is a written, brief description used to describe the project to

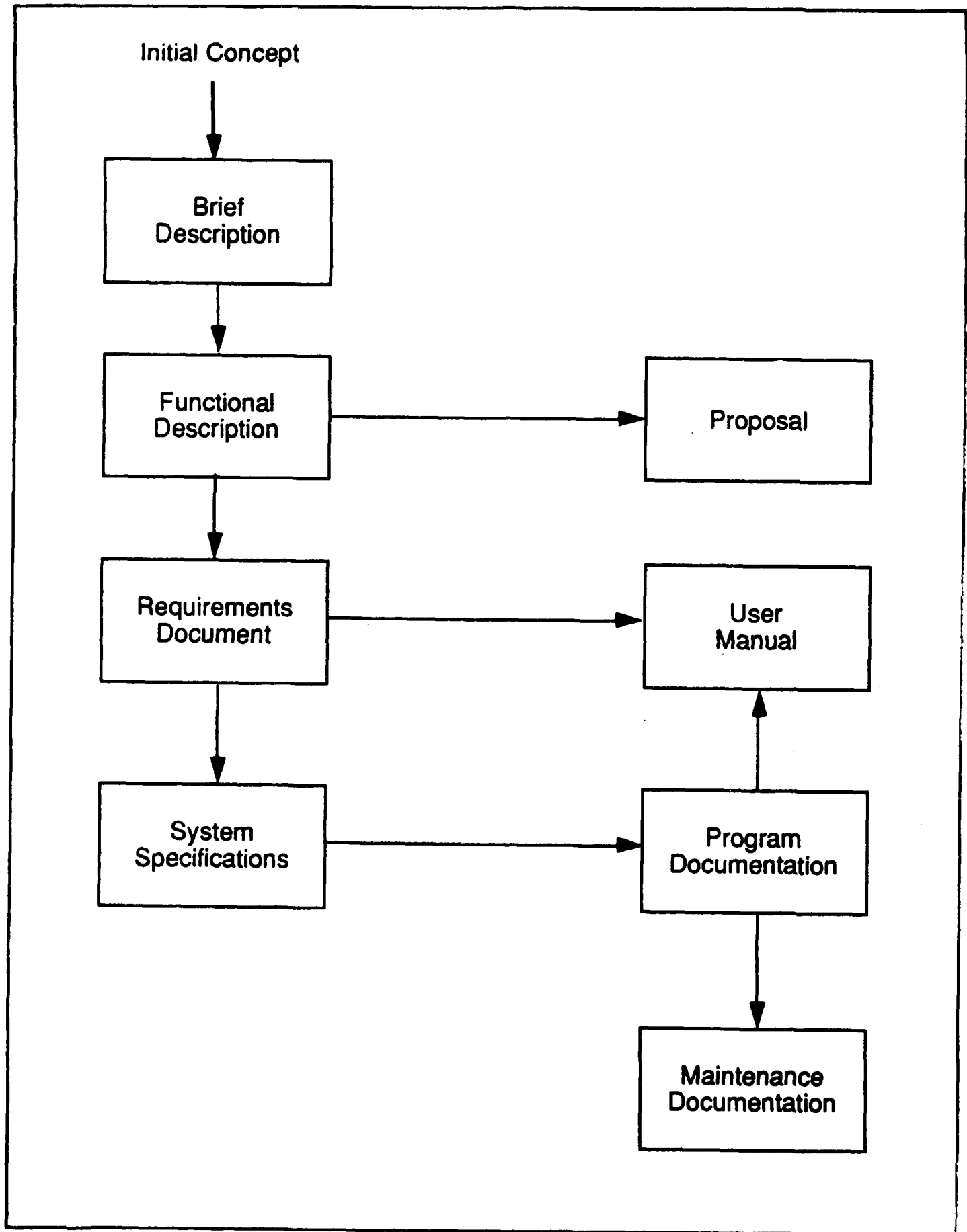


Figure 11-1 Expert System Documents

potential sponsors and to evaluate the project for its benefit to the organization.

High level synthesis is a stage in the initial concept design where major decisions about the scope and the basic nature of the expert system are still being made. At the beginning of high level synthesis, the most basic questions of scope are considered. Who will use the proposed expert system? What decisions will be supported? What data will be provided, and which data will be automatically accessed and which provided by the user at run time? What information will be output, to whom, and for what use? Even after the initial concept is documented in a brief description much high level synthesis work will remain. As an example, consider the MOBPLEX expert system. Its brief description is given in Figure 11-2. This brief description, along with brief descriptions of several other potential expert systems, was subjected to a screening and selection process as described in the project initiation chapter. After the MOBPLEX expert system was selected for development, the conceptual design phase began in earnest and many basic features of the expert system began to take shape for the first time. The brief description therefore is a decision document, not a document which provides a full functional or requirements definition.

There are several tools that might be utilized in preparing a high level synthesis so that it can be written into a brief description. Two major tools are lists or taxonomies of attributes of the expert system and analogy. Both of these tools will be discussed in the following paragraphs.

Lists or taxonomies are good tools to use during high level synthesis. These lists list the characteristics or attributes that the expert system should address and perform, or in some cases, input and output of the system. For example, Figure 11-3 shows a

MOBPLEX Project Description

The MOBPLEX project is to build a prototype expert system to support Mobilization and Deployment Planning and Execution for the U.S. Forces Command (FORSCOM) at Ft. McPherson, GA. MOBPLEX will support J5 Mobilization Planning and Mobilization Stations located throughout the country. The system will use an object oriented user interface to allow mobilization planners to access various databases:

- the PTSR submissions by each Reserve and National Guard unit scheduled under full mobilization,
- the most recent Mobilization Station shortfall reports,
- the relative priority of the units for mobilization,
- the estimate of the number of days required for each unit to process and train at the mobilization station, and
- the estimate of travel time from home station to mobilization station following mobilization orders.

The system will encapsulate knowledge in an expert system about the meta-data of the databases, thus allowing the user to manipulate multiple databases with no knowledge of the meta-data.

The system will have the capability of evaluating each unit's PTSR submission for coherence of data included in the PTSR in a manner similar to the way an experienced, trained human evaluator would.

Based on the data submitted in the PTSR, the system will schedule the various units through POM/POR processing, various firing ranges, various maneuver areas, and required transportation resources. The output reports will also provide insight into facilities loading, training Class III (POL) requirements, training Class V (ammunition) requirements, and other training logistical support requirements.

Figure 11-2. MOBPLEX Brief Description

OBJECTIVES:

The primary objective of the MOBPLEX system will be to provide decision support for mobilization planning at the MOBSTA, CONUSA and FORSCOM. These will be accomplished using the following:

- Increase timeliness of PTSR data
- Make PTSR data more consistent. Create a PTSR checker that will automatically check the data for internal consistency using the expertise of a MOB planner.
- Provide scheduling support for "what if" modelling and to reduce the work load of the MOBSTA mobilization planner when time lines change.
- Maintain PTSR database on PC. Provide for electronic transfer of DARMS data to MOBPLEX using simple interface running in the available Z248 at the MOBSTA.
- Provide simple user interface: minimize typing, navigate databases easily, give various perspectives of the data, and use Lotus-123 for ad hoc report generator.
- Provide standard reports. The standard reports to be printed from the system.

Figure 11-3. Extract of MOBPLEX Attributes List

list that was made for the MOBPLEX expert system, providing attributes and functions that the system should perform. These lists were created during the initial conceptual design phase of MOBPLEX based upon information from current FORSCOM practices. The information was derived from a survey of MOB planners and the experience of the J6 in building previous systems. Expert systems literature was also researched to determine what technological tools were available for database access support using expert systems. The list attempted to determine all the uses that might be required of a mobilization expert system.

Analogies are sometimes a powerful way to synthesize a concept of what an expert system should be. An analogy forms a mental picture called a metaphor that allows readers of the brief description to understand the complete overview of the functionality of the expert system.

The brief description should be a written document no more than two pages in length that fully develops the synthesized ideas and concepts for the expert system. Any precise boundary between the high level conceptual design and a more detailed design of a functional description is somewhat arbitrary, but there always comes a time when the expert system designer has settled on the basic concepts of what the decision problem is, who has it, what data are appropriate and available, what models are appropriate and available, and what computer system and data communication systems are appropriate and available. All of these things can easily go into a brief description. After the brief description is completed and the decision has been made to select a particular expert system, the design sessions become more detailed and the structure begins to be more definitive. At this point, the process of functional design and functional requirements has begun. This will

provide information about what the user will actually do in an expert system session, and what the system will do in response. This is the point where high level synthesis and conceptual design ends and detailed synthesis in the requirements definition begins.

11.4 Functional description of an expert system.

The following general format is recommended for functional descriptions of proposed systems that constitute complete expert systems for an ongoing operation in which the same decisions are already made routinely.

1. Title.
2. Job Description. Functional description of the responsibilities of the office or person who performs the task that is proposed to be encapsulated in the expert system. If the task to be supported is one of a group of related tasks, that group should be described here.
3. Task Description. Detailed description of the task that the proposed expert system will assist.
4. Information Environment and Resources. Detailed description of the data bases, communication systems, and computer systems in the immediate area of the proposed expert system; specifications of the equipment on which the system is to run, if that is fixed in advance of solution definition; list of the data sources available to the proposed system if data bases in the immediate area are not the only sources; architecture of the immediate information environment. If the proposed system is one of several alternatives this section may be omitted and the information environment and resources may be described elsewhere for the whole spectrum of alternatives.
5. Expected enhancement. Point by point comparison of the proposed way of handling each sub-task in the task with current procedures. This is usually written in such a way as

to contain implicitly a list of benefits and to provide a framework for benefit estimation. If the task is not currently being routinely performed, this section can list the benefits of performing the task, compared to current methods.

6. Operators. Who or what group will operate the system in the sense of using it.
7. Existing procedure. How the function is now performed step by step. For some systems this section may be redundant for section 5. But the rationale for this format is that it is often easier to understand comparative explanations. The benefits can be discussed in a comparative way which immediately tells the reader why the proposed system is of interest and explains what it does by comparison to familiar ways of doing the same thing.
8. Key concepts. The unique, clever, elegant, original or creative ideas that make the system viable are explained in this section. Anything unusual about the proposed expert system is mentioned.
9. Inputs. There must be a complete list of data items needed to feed the system, and for each item there must be listed a normal source, an informal definition, and alternative source if available, a declaration of committed values, a default value, and also any special requirements regarding data quality, integrity, age, user overrides, etc.
10. Outputs. All outputs are listed and described including the purpose, destination and intended uses of each output item as well as its information content.
11. Benchmark sessions. At least one typical interactive session with the proposed expert system should have been envisioned in enough detail to allow documentation here of at least a narrative description of a typical use of the system. The description would give the goals of a typical session, the data state upon session initiation, a broad narrative of how the user would interact with the system, and what output would

be produced.

12. User interface. This section narrates how the operator interacts with the system including the most important interface protocols for collecting user input and for generating reports.
13. Equipment and utility software. If section 4 above is omitted or is written in general terms, the equipment constituting the work station for the proposed system is listed here. Requirements for utility software are listed here if they are not obvious from the environment description.
14. Deliverables. A list of what hardware, software, documentation and other items that would constitute a delivery of the proposed system.
15. Estimate of development effort and or development plan. If the aim of this detailed description is evaluation of the proposed system against a few alternatives, an estimate of effort needed to develop the system is given here. If the aim is to proceed with design, a development plan and schedule should be reported here.
16. Estimate of the effort required by the domain experts assisting in designing and building the system.
17. Estimate of testing and evaluation effort and or testing and evaluation plan. If the aim of this functional description is evaluation of the proposed system against a few alternatives an estimate of the effort needed to test and evaluate the developed system is given here. If the aim is to proceed with design, an evaluation plan should be reported here. This might include a plan for user testing, for expert peer testing, and verification of the system scope. If life cycle costing methodology is being used for evaluating this system, a preliminary life cycle cost profile should be reported here.

11.5 The Proposal.

When writing an expert systems proposal, you must clearly understand that technology itself is not a reason for implementing a system. A table of contents of a typical proposal is shown in Figure 11-4. The expert system is a tool to be used in order to insure benefits by increasing productivity, processing paper work faster, and/or serving customers better. Simply investing in expert systems in the hope of increasing productivity will not necessarily get the intended result.

The expert system must be founded upon clear business benefits or requirements of the organization. Do not allow the technology issues to jump ahead of the organizational benefits. The expert systems proposal is intended to make a fundamental change in the way a company does business, so do not present it as a new technology. In creating a proposal, the champion must take a strong leadership role. He needs to decide whether the expert systems are needed either to increase or to maintain organizational productivity. This answer can come only from the line operating units and not from MIS groups.

The brief description and the functional description that have been prepared form a foundation for the proposal. A proposal should contain three basic elements: introduction and background, main body, and cost benefit analysis. The introduction and background explains why a potential sponsor should be supportive of the project. In addition, the introduction and background should include the rationale for the incentive for the proposed system. It should include the environment in which the system will operate and a short statement of what the proposed system is and generally how or why it will alleviate the difficulties identified in the environmental statement.

Title of Proposal

Summary

Table of Contents

1. Introduction and Background

1.1 Why

1.2 What

1.3 Who

1.4 How

1.5 Where

1.6 When.

2. Proposal

2.1 What you propose to do.

2.2 Deliverables for the project.

2.3 Initial work plan for the project. Estimate of time and effort.

3. Benefits and Costs

3.1 Estimate of Value of implementing system to user organization.

3.2 Estimate of value to sponsor.

3.3 Initial budget estimate for cost of project. Include full life cycle costs.

Figure 11-4 Title Page For Proposal

The main body of the proposal defines precisely what you propose to do. It defines the scope of the system and its progress towards development of the complete expert system environment. The proposal should outline all the deliverables of the system. It should also define the technological scope of the system specifically stating what kind of prototypes and how many will be delivered. The expected complete functionality of the final product should also be forecast. The proposal should also have an implementation plan included. (Implementation considerations are contained in Chapter 10 of this guide). The implementation plan defines milestones and an estimate of the development schedule. The third important element of the proposal provides the costs and benefits of the system. (A guide to the preparation of impact assessments and cost benefit analysis are contained in Volume 2 of this guide.) In the proposal, the result of the impact assessment and the cost benefit analysis should be summarized in order to define the benefits of the final system to the user organization and an estimate of the cost of developing the system as well as its ongoing maintenance cost for the life of the project. A specific budget for the proposed project should be given in this section giving a work breakdown structure that conforms to the potential sponsor guidelines.

Appendices to the report can also be included. Particular appendices might be brief descriptions, the functional description, and qualifications of the proposed team to accomplish the work.

11.6 Requirements Definition Document.

A Requirements Definition Document contains the results of functional design and describes the behavior of an expert system in sufficient detail to begin building the expert system. In the iterative development methodology, the requirements definition becomes the primary document to guide the development since

specifications are not prepared in as much detail as in the traditional development methodology. The iterative development assumes the user cannot define his total requirements until the prototyping phase begins. As the user becomes familiar with the capabilities of expert systems, he will find more requirements for the system that he initially did not consider. The requirements definition then should focus on scope of the project rather than specific details.

If the system is to be implemented by a contractor or agency that is separate from the designer, a formal Requirements Definition Document is essential. Also, if significant parts of the hardware and software resources remain undecided at the time of functional design, a Requirements Definition Document is useful because it allows the design to proceed as far as it can while remaining hardware-independent and software-independent. If a formal Requirements Definition Document is prepared, it documents functional design and provides the starting point for program design.

The Requirements Definition Document gives details of the structure and behavior of the major modules and of their interfaces with each other and with the outside world -- the user, communications links, interactions with databases and other systems, etc. Each major module is left as a "black box" whose inputs, outputs and behavior are completely specified in the Requirements Definition Document but whose internal workings are left to the program designer.

Unlike specifications, requirements can be independent of the hardware, independent of the utility software and programming languages, and independent of the specific implementation of database management. Requirements are not normally independent of the formats of piped-in data from outside sources; since the program designer is assumed to have no control of formats of

outside inputs, the requirements should usually require extensive modularization (isolation) of data inputs. If there are unanswered questions as to what incoming data will look like, the Requirements Definition Documents should require a data interpretation module whose output but not input is well defined; later the solution definition must define not only the input but also the details of conversion to the output (that constitutes input to the rest of the expert system).

A recommended format for Requirements Definition Documents is given in Figure 11-5. This format assumes that the functional design has broken the system down into separate modules, each of whose requirements can be considered separately.

11.7 Solution Definition.

A Requirements Definition Document and definitions of incoming data are the major elements needed to begin solution definition. The solution definition provides exact choices of all resources (hardware, software, utilities) and an overview system specification. The expert system designer's main job is to produce a solution definition that satisfies all of the requirements in the definition and provides the initial guidance for the design and prototyping phases of the methodology.

Detail program specifications are not usually prepared for expert systems. But a system specification that provides general guidance for the development is required. Program design is generally done simultaneously with the prototyping phase. The system specification must be revised during the development to reflect the current state of the design. Thus, the expert system prototype becomes a "living specification" of the system.

11.8 Program Documentation.

Program documentation is delivered in the form of the Program Maintenance Manual and the soft copy documentation to which it

Requirements Definition Document

1. Title and Background. Use summary of the functional description.
2. Architecture of Major System Modules. Provide a modular breakdown of the system. Names and short descriptions of the modules that are anticipated in the system. This should also include data links to other system when required. This is an overview section intended to orient the reader into the uses of the system. Full module descriptions are given in Section 3.
3. Descriptions of Module Requirements. Each major module is described with its requirements.
4. Definitions of Major Variables and Parameters. This section describes and defines the data elements required in the system. If some data must come from other systems, this is the section to describe when and how the data must be imported. This will provide a glossary of terms for the solution definition.
5. Model Descriptions. If any models are already in use in either a manual process or automated process, these models should be evaluated for inclusion in the expert system. Expert systems usually include algorithmic components.

Figure 11-5. Requirements Definition

refers. This manual is a hard copy document which must describe the expert system in sufficient detail to provide for installation, testing, maintenance and enhancement of the system. It delegates part of this responsibility to soft copy, but its organization and table of contents should be complete, as if everything except the detailed source code were printed in the manual itself. The soft copy portion of program documentation consists of the documentation file and the source code.

11.8.1 Program Maintenance Manual. The Program Maintenance Manual provides the technical information required by programmers for enhancing the expert system. You must document the interactive interface of the expert system in detail. This is best done through screen documentation and logic flow charting of the whole system. If your expert system is written in a highly structured language such as Ada, logic flow charting should be augmented by structured code or pseudo-code where necessary.

The Program Maintenance Manual, together with the documentation file and the source code, will be used for maintenance and enhancement. This means that computer-competent personnel must be able to use this program documentation effectively to add or modify any of the expert system's features. This is a severe test of program documentation. Furthermore, the requirement given maintenance and enhancement personnel will normally be functional requirements ("Make the system able to ..."), so those future personnel will need to understand the intended purposes of user interactions -- that is, to understand why the system behaves rather than simply how it behaves. It is not enough to say, "Let them also read the User Manual." The Program Maintenance Manual should tie requirements to design. It should not simply be an abridgment of the User Manual, but should treat the requirements from the system provider's viewpoint rather than from the user's

viewpoint. This means that each requirement description should refer to the design features that implement it.

Data and communication resources are in another category of maintenance documentation. Generally the Program Maintenance Manual for an expert system that performs data extraction should include:

- * A brief description of each database from which the expert system will extract data, including the correct official name of the database, who is currently responsible for maintaining its data contents, who is responsible for its formats and definitions, and a reference to its official documentation.
- * A brief description of each file, report or grouping of data that the expert system will extract from the database.
- * A brief description of how extraction is handled (even if extraction is automatic), including current access codes and formats and current network access codes or how these are determined in the field.
- * A set of references to system documentation to identify where changes need to be made if there are changes in the access codes, formats, protocols or structures of any of the extracted databases or of any of the communication channels to them.

Even trivial changes in extracted databases or in communication channels will cause malfunctions in data base extraction. The documentation must explicitly define all data base connections to facilitate maintenance of the data base extraction.

11.8.2 Source Code -- Internal Documentation. The expert system shells have evolved so that their programs are largely self-

documenting. This is especially true of rule based systems. Objects are not always as transparent as rules and may require more extensive internal documentation than rules.

Naming of variables, commands, subroutines, functions, files and utilities is an important part of internal documentation. With good descriptive names, the shell code becomes easy to read and understand. Remember that the shell only provides a syntax; you provide the vocabulary. Besides naming, there are other elements of style that enhance internal documentation. These include indentation, punctuation, and rules for transfer of control.

11.8.3 Documentation File. The Documentation file can be a separate file or it may precede the main program file. It is written in comments of the expert system shell language or an ASCII file. Nearly everything in the documentation file should be printed in the hard copy system manual as well, but it is important that such things as module directories and glossaries of variables be in a file so that the maintenance and enhancement personnel can use text editing or query utilities to find names and locate lines of code where a given name appears.

The Documentation file should contain as a minimum the following:

- * Definition of all global variables or those common to more than one function.
- * Definitions of the purpose and operation of each module of the system.
- * A glossary of all variable names, modules, chunks, utility names and file names.
- * Technical on module operation and function
- * The date and time of the latest revision of the documentation file.

On-line user documentation, to be discussed along with other user documentation in the next section, is of interest to maintenance

and enhancement personnel as well as to users. Menus, command lists, error messages and HELP messages should be available either in source code or in the documentation file, so that maintenance and enhancement personnel can make automated searches through them.

11.9 User Documentation.

User documentation in the form of the User Manual is one of the most important interfaces the user will have with the system. A successful expert system may have hundreds of users and be used thousands of times. Because development effort is amortized over so many sessions, it makes practical sense to invest in great sophistication to achieve simplicity and efficiency for the user. The user is not expected to be a computer programmer, and is not expected to read system documentation. There is a separate category of documentation for the user.

User documentation consists of three sources of information to inform the user: intrinsic documentation consisting of on-screen instructions, menus, screen layouts and general transparency of the interactive interface; a hard copy User Manual; and on-line documentation, including HELP if provided.

11.9.1 Intrinsic Documentation. There are some categories of user documentation that do not require coverage in the User Manual or in on-line documentation. These categories of intrinsic documentation include everything that the user can reasonably be expected to know already, or to find out from a source other than formal user documentation. You should review these categories before deciding what to include in the User Manual and on-line documentation. While reviewing intrinsic documentation, tie down its sources. The User Manual has ultimate responsibility to give the user all required information; it can delegate the presentation of some categories to other media, but delegation must be specific, so the reader of the User Manual can tell exactly where he or she

must go to find out each thing not explicitly given in the manual.

Hardware and software resources are in another category of intrinsic user documentation. For example, the MOBPLEX User Manual is written assuming that User Manuals for the computer, the operating system and the mouse are readily at hand. A User Manual should not quote or paraphrase at length from existing manuals for imbedded software utilities such as a DBMS or graphics driver. On the other hand, a balance must be struck; in general, the User Manual should specifically include anything that can be put on a single page, and anything that involves choices of alternatives by the user.

Any data base extractions for the expert system must be documented in the user documentation. If the designer has provided user-specified access codes or interactive extraction procedures, these would be explicitly put in the User Manual. The following items should be in the User Manual:

- * A brief description of each file, report or grouping of data that the expert system will extract from the database.
- * A brief description of how extraction is handled (even if extraction is automatic), including current access codes and formats and current network access codes or how these are determined in the field.

The interactive interface itself is a fertile source of intrinsic documentation. Ideally, the user should be able to operate the expert system without needing to refer to either the User Manual or to the on-line documentation. The screen layouts, on-screen instructions, and menus can make it obvious to the user how to proceed. The protocols, if consistent enough and if well enough based on good enough metaphors, can allow the user to infer from

having done one thing -- how to do another thing.

The on-screen instructions and menus are documentation in a real sense. Their syntax gives the interactive interface a personality; their vocabulary gives the designer powerful control over some aspects of the user's mental processes while interacting with the expert system. The syntax should be as simple and consistent as possible. Unlike a person, whose personality should be interesting, an interactive interface should have a transparent personality, which is one that is, above all, predictable. A good interface should be dull, flat and boring if the user focuses on the interface itself, because the idea is for the user to focus through the interface on the problem being solved. The user should not be thinking of moving a mouse and pressing one of its buttons, but of selecting or deselecting an object; and the user should not think of that object as a menu item or a picture, but as the real world object it represents.

The vocabulary of on-screen instructions and menus does two things: it keeps the user focused on the problem rather than the interface, and it connects parts of the interface together. Always use words that describe real-world entities and actions, rather than computer entities and actions. As an example, if an expert system is collecting input from a user, you would never see the word "collect" in instructions or menus; even the word "enter" is not as good as a situation in which the entering is an obvious requirement and the focus is on the objects themselves.

One rule regarding vocabulary of on-screen instructions and menus is to avoid synonyms. There is usually one best name for an object, and that one name should be used everywhere. Consistency of vocabulary throughout the system is very important to the ease of use of the system. The most common exception to this rule is to use class membership synonym. For example, an "Engineer

Detachment" is one kind of "mobilizing unit," so a static menu item or instruction might mention "mobilizing unit" and be valid whether the mobilizing unit is an Engineer Detachment or something else. However, in many such cases it might be better to make the menu item or instruction a dynamic one, so that Engineer Detachment (or whatever the object is) would be dynamically inserted into the menus based on the system state.

Another rule for on-screen instructions and menus is to avoid forced reading. Suppose you are ready to enter something, and you know its proper format and content. At that moment, your only interest in the display is to see whether the system is ready to accept this particular input. What you want to see is the name of the kind of data you are ready to enter. If this name is displayed, but accompanied with an explanation telling you all the legal ways to enter it, then you must at least glance at this material. If there is too much of it, it will interfere with your ability to spot the looked for name, and it will interrupt you while you inspect it. Explanatory information should be placed in pop-up windows available on demand from the user.

The interactive interface must provide intrinsic documentation for both novice and expert users. One way of serving both is to relegate lengthy instructions to on-line documentation such as HELP messages. Another is to put instructions in a special place away from the area that solicits an input. The novice user can enter HELP or glance at the explanation area, while the expert user can go ahead unhindered. For example, MOBPLEX maintains information about system state in a box in the lower right hand corner of the screen. The main menus are always across the top of the screen with associated pop-up windows for choices. Data input is into pop-up windows containing forms in the main window of the screen. Most input is quickly accomplished with a mouse rather than keyboarding.

Pop-up menus have the advantage of avoiding screen clutter and displaying information only when it is needed. They are certainly better than using separate screens for each subfunction, because the remainder of the display not covered by a pop-up serves to keep the user oriented. Keeping the user oriented to his location in the system is important to the ease of use of the system.

Whether subfunctions are handled by pop-up menus, by pop-up windows that handle a subfunction in a more complex manner than simply by menu choice, or by separate screens it is important to realize that the architecture of the organization of subfunctions is a form of intrinsic documentation.

The wrong way to organize subfunctions is just let them evolve. Suppose the designer fails to heed advice given in Chapter 9 and simply proceeds as follows: while designing a function, this designer encounters a subfunction that is complex enough to deserve its own pop-up menu or pop-up subfunction window (or, even worse, a separate screen). Then, inevitably, the same designer will find a subfunction within that subfunction that deserves the same treatment. Unfortunately, many interactive interfaces are designed along these lines.

A subfunction architecture that simply evolves can become a documentation nightmare. There is never any real excuse for a screen architecture to have more than three or four levels. If the user can be in a screen within a screen, the user can get lost and must be given some landmarks. You may have to devise after-the-fact names for sub-sub-...functions (if a screen is reached by menu selection, use the menu item as the screen name) and label each screen with the name of its parent screen (or with its entire ancestry).

Well-constructed subfunction architecture gives an expert system a user-understandable structure and thus constitutes intrinsic documentation. Other intrinsic documentation coming from good interface design includes screen layout consistency and protocol consistency. When the same information is in the same part of every screen, its location on the screen helps the user understand it. Thus, in MOBPlex the lower right corner of the screen always carries information about the user's location in the hierarchy of the data base levels. In menu-driven systems, designers should always put the menu items in a consistent order, such as with the most typical function at the top and "exit to previous screen" at the bottom. Protocols would allow the user to predict the rules for entering one datum from the files for others.

11.9.2 User Manual. No matter how easy the system is to use, a User Manual is a required component of any expert system. The User Manual does not replicate other documentation but provides a guidebook on how to use the expert system and a reference to details and use of each function provided by expert system. The general flow of the manual should include the following.

1. Introduce the expert system to the reader to give enough information to determine if the expert system is applicable to his/her problem.
2. The next section aids the user in Getting Started with initial hardware and software configuration and system initialization.
3. Learning the system provides for an overview of the expert system environment and concludes with a step-by-step example to walk a first time user through the expert system functions.
4. System References contain a detailed description of each function and its use within the expert system.

The operator consults user documentation:

- * to learn generally about the expert system
- * to ascertain the applicability of the expert system to a problem or class of problems
- * to find out how or whether the expert system treats certain aspects of a problem
- * to learn how to accomplish a particular high-level function
- * to refer to details of an interactive procedure

The INTRODUCTION section should provide the reader with a concise overall description of the expert system. This allows the user to determine if the expert system has application to their problem. Putting this first in the introduction saves the user from having to search the manual to make this determination.

GETTING STARTED begins by the user determining if he/she have the necessary hardware and software before attempting to set up or install the expert system. This section may also have a "fast track" option for the experienced user. The hardware configuration section should include: type or class of computer, necessary and suggested peripherals, minimum memory, and minimum floppy and/or hard disk storage required. Software configuration section should include: intended operating system or systems, and any additional support software.

The installation section must contain a clear step-by-step process the user can follow explicitly. If available, set off the commands the user is to perform in boldface. Having the directions in normal typeface and the commands in bold type allows the user to know at a glance what they are to type or to perform.

The SYSTEM OVERVIEW should provide the user with a brief overview of how the expert system serves the purpose stated in the

INTRODUCTION. This overview is followed by a detailed description of each component of the expert system. The overview section should give the user an overall conceptual view of the expert system.

With a rough understanding of how the expert system works, the user is then prepared for a step-by-step example of solving the problems the expert system has been designed for. This example should be as simple as possible while including essential features necessary to solve actual problems. The example should walk the user through the process step-by-step, explaining to the user what he is doing at each step. For good implementation strategy, it is essential the user understand why he must do a particular step.

The User Manual must document each feature completely, regardless of whether or not he/she is able to envision everything the user can do with it. Users always find shortcuts, and they find roundabout ways of accomplishing functions that the designer did not anticipate. For many complex and flexible expert systems, it is simply impossible for the designer to predict all the different ways that users could combine features to accomplish aims.

The SYSTEM REFERENCE section provides a detailed description of each feature in the expert system. These references should be grouped together into components by screen or by the logical organization of the expert system.

In the User Manual for MOBPlex the explanations are centered around each screen and the screens themselves are structured in a logical flow. Within each screen in the MOBPlex expert system every user action that can be executed from that screen is fully described including the interactions. This includes underlying assumptions both before and after each user action. Furthermore, any underlying algorithms and features should be explained. Logical

flow in MOBPLEX is centered around operations and the actions within operations. The MOBPLEX User Manual starts with a detailed explanation of the operation screen followed by a detailed explanation of each action screen.

The carefully organized User Manual can thus serve both reference and tutorial purposes. Its overall organization is by very-high-level functions that are not only understandable to the novice user but also classify the features of the expert system into separate groups.

A good index is essential. No matter how well the manual is organized the user can always ask a question whose answer is found in a combination of places. A useful optional feature of an index is the key reference for each entry, which is set off from other references by boldface or other means. The key reference is the one that defines the entry rather than simply mentioning it.

A glossary is usually needed. It should contain definitions of all expert system-specific words and phrases. It may also contain definitions of ordinary words, but if the manual has too many "big" words in it, you should usually substitute a simpler word rather than generate a glossary entry. However, if a "big" word is used many times in the manual and no simpler word carries the exact meaning, put it in the glossary.

A good User Manual should include helpful hints and warnings to assist in avoiding problems in the operation of the system for hardware software and the required data bases. They should be set apart from other text and easily identifiable.

11.9.3 On-Line Documentation. Some expert systems provide specific on-line documentation as part of the shell. There is usually no reason the user should not consult the hard copy User

Manual while operating, so on-line documentation should not be provided without first checking to see if there is any real need for it.

For most expert systems the best form of on-line documentation is dynamic HELP. The main idea of dynamic HELP is that a user can enter "HELP" (or select a HELP menu item) in the middle of a procedure, and the expert system can consult its state to see what help the user needs rather than ask the user to specify something that the user doesn't know. Using ordinary static HELP systems is like trying to use a dictionary to look up a word that you have no idea how to spell. The user can, for example, activate a function or object that may or may not be the intended one, then enter HELP and receive a message specific to the active function or object.

The main disadvantage of dynamic HELP is its difficulty of implementation. You need to add an entire state-classification structure to the logic. Each message that is provided must be activated if and only if its particular state exists. It probably takes at least 200 messages to comprise a good dynamic HELP system for a typical expert system. Such a system still falls short of being a complete substitute for the hard copy User Manual, because although it can document each interactive protocol, it cannot explain modelling concepts and high-level functions adequately.

INDEX

Index

algorithmic	104, 146
backward chaining	25, 64, 66, 68, 74, 77, 78
categorize your problem	iii, 65
compilation	64
constraint satisfaction	74, 76
control windows	117
cost benefit	5, 33, 49, 141, 143
delivery platform	iii, 33, 64, 69, 71
design problems	iii, 35, 67, 68
development interface	iii, 64, 69
development platform	iii, 64, 71, 72
diagnostic problems	iii, 66, 68
display layouts	iv, 109, 114
documentation . . iv, v, vi, 2, 4, 36, 81, 82, 86, 131, 132, 139, 140, 145, 147-156, 158, 159	
documentation review	iv, 86
domain expert . . 34, 35, 48, 52, 55-57, 59, 60, 63, 86-89, 91-98, 123, 124, 127	
end user interface	68
estimate	48-52, 140, 143
evaluation	3-5, 27, 34, 36, 52, 53, 56, 94, 124, 140
expert system design	iv, 107
expert system shell . iii, 4, 16, 21, 55, 64, 66, 71, 76, 78, 149	
fact base	23, 24
forward chaining	25, 66, 74, 77, 78
functional description . . v, 131, 137, 138, 140, 141, 143, 146	
hypothetical reasoning	68, 74, 77, 81
if/then	24, 25, 86
impact assessment	5, 143
inference engine	21, 24, 25, 26

Index

interface requirements	iii, 64, 65, 68
interpret	21, 76, 80, 92
interpreter	21
interview	58, 87-93, 105
interviewing	iv, 58, 86, 87, 89-91
knowledge acquisition	iv, 31, 33-35, 50-53, 59, 60, 85-87, 90, 91, 94, 96-98, 123, 124, 127
knowledge base	ii, 5, 17, 18, 24-27, 36, 70, 78, 85, 86, 121
knowledge base validation	5
knowledge engineer	34-36, 49, 55-61, 63, 86-93, 95-97, 123
knowledge engineering	4, 49, 58, 61, 85, 86, 93
knowledge modelling	iv, 98, 100, 105, 107
knowledge representation	64, 92, 98, 99, 102
message windows	117
method of reasoning	64
modelling	iv, 98, 100, 105-107, 121, 136, 159
monitoring	iii, 66, 67
narratives	iv, 94, 95
object oriented programming	76-78, 81
on-line	vi, 68, 149-151, 153, 158, 159
one page description	31
perceived objects	iv, 107-109
pilot project	31, 33, 48, 49
planning problems	iii, 68
platform requirements	iii, 65, 71
pop-up	115-117, 119, 153, 154
problem type	iii, 64, 65
process control	67
program documentation	v, 145, 147
proposal	v, 11, 141-143
protocols	iv, 86, 94, 96, 97, 140, 148, 151, 155

Index

prototyping phase v, 34, 38, 52, 53, 114, 117, 123, 124, 144, 145
 quality 36-38, 44, 57, 89, 139
 quality assurance 36-38
 requirements definition, 31, 33, 35, 38, 50-52, 134, 138, 143-146
 rule base 21, 24
 rule based programming 74, 77, 78
 scheduling problems iii, 67, 68
 shell capabilities iii, 72, 74
 shell selection iii, 64, 65
 simulation iv, 69, 86, 96, 97
 specification 33, 35, 117, 145
 stakeholder 56, 59, 125
 system interfaces iii, 70
 system specification 145
 truth maintenance 68, 74, 76, 77
 user iii, iv, vi, 1, 4, 15, 18, 19, 23, 24, 29, 34, 36, 53, 55,
 57-59, 64, 66-70, 74, 76, 77, 80, 85, 96, 105,
 107-109, 113, 115-117, 120, 125, 129, 130, 131, 132,
 134, 136, 137, 139, 140, 143, 144, 147, 149-159
 user interface . iii, 4, 15, 34, 64, 68, 69, 85, 107, 108, 117,
 136, 140
 User Manual vi, 109, 131, 147, 150, 151, 155, 157-159
 validation 3, 5, 36, 38, 53, 105
 vendor support iv, 64, 81
 verification 3, 89, 105, 140
 windows 108, 109, 114-117, 153, 154
 Z248 136